OBSTACLE AVOIDANCE PATH PLANNING BY THE EXTENDED VGRAPH ALGORITHM

NAGW-1333

By:

C.H. Chung
G.N. Saridis

Department of Electrical, Computer and Systems Engineering
Department of Mechanical Engineering, Aeronautical Engineering & Mechanics
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

CIRSSE Document #12

OBSTACLE AVOIDANCE PATH PLANNING BY THE EXTENDED VGRAPH ALGORITHM

by

C. H. Chung and G. N. Saridis

Robotics and Automation Laboratory
Department of Electrical, Computer and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

January 1989

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In many path planning algorithms, attempts are made to optimize the path between the *start* and the *goal* in terms of *Euclidean* distance. Since the moving object is shrunk to a point in the *Configuration Space*, *Findpath* can be formulated as a graph searching problem. This is known as the *VGraph Algorithm*.

Lozano-Pérez points out the drawbacks of the *VGraph Algorithm*. The first drawback is related with rotation of a moving object. This drawback can be solved by using the *sliced projection method*. However, the *VGraph Algorithm* has serious drawbacks when the obstacles are three-dimensional. The *Extended VGraph Algorithm* is proposed to solve the drawbacks of the *VGraph Algorithm* by using the *Recursive Compensation Algorithm*. The *Recursive Compensation Algorithm* is proposed to find the collision-free shortest path in 3D and it is proved to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*.

1. INTRODUCTION

1.1 Motivation

In many path planning algorithms, attempts are made to optimize the path between the start and the goal in terms of *Euclidean* distance. In the *Configuration Space* [9], the moving object is shrunk to a *Configuration Point*, while the stationary obstacles are expanded to fill all space where the presence of the *Configuration Point* would imply a collision of the object with obstacles. Therefore, *Findpath* [29] can be formulated as a graph searching problem. The graph is formed by connecting all pairs of visible vertices of the *Configuration Space Obstacles*.

Consider the *VGraph Algorithm* for a moving object to find the collision-free shortest path in a workspace with some obstacles. A lot of work has been done in this field, which has the following design steps;

- Build the *Grown Space Obstacles*.
- Find the visible vertices by detecting interferences.
- Build the *VGraph* with a set of the visible vertices.
- Search the *VGraph* by the graph search algorithm.

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez [29] points out the drawbacks of the *VGraph Algorithm*. The first drawback is related with the rotation of a moving object. Since the *VGraph Algorithm* require moving an object along obstacle boundaries, shortest paths are very susceptible to inaccuracies in the object models. This drawback can be solved by using the *sliced projection method* [28] [29] [30]. However, the *VGraph Algorithm* has serious drawbacks [29] when the obstacles are three-dimensional:

- shortest paths do not typically traverse the vertices of the *Grown Space Obstacles*,
- there may be no paths via vertices, within the enclosing polyhedral region R, although other types of safe paths within R may exist.

Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the

Grown Space Obstacles. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. So, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D. Therefore, a new algorithm, called the *Extended VGraph Algorithm*, should deal with the drawbacks of the *VGraph Algorithm*.

The **Extended VGraph Algorithm** has the following design steps;

1) Apply the *Orthogonal Projection Method* to get the *Grown Space Obstacles* in 3D.
   i) Project obstacles in 3D onto the projection spaces.
   ii) Build the Grown Space Obstacles in 2D.
   iii) Select the necessary *Grown Space Obstacles* for the *VGraph*.
   iv) Reconstruct the Grown Space Obstacles in 3D.
2) Find the visible vertices by detecing interferences.
3) Build the *VGraph* with a set of the visible vertices.
4) Search the *VGraph* by the graph search algorithm.
5) Apply the *Recursive Compensation Algorithm* to obtain the collision-free shortest path in 3D.

The following results have been obtained by the *Extended VGraph Algorithm;*

- The *Extended VGraph Algorithm* can deal with not only translations of a moving object but also its rotations by using the θ *sliced projection method.*
- Since the *Orthogonal Projection Method* avoids building the unnecessary *Grown Space Obstacles*, it can make the *VGraph* simpler than any other algorithms that use all of the *Grown Space Obstacles*. Therefore, the *Orthogonal Projection Method* can save the memory space to store the representation of the *Grown Space Obstacles* and it can shorten the graph search time because of the simpler *VGraph*.

- The *Recursive Compensation Algorithm* can guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*. The property of convergency of the *Recursive Compensation Algorithm* is proved. Since ϵ is set to 10^{-5}, Lozano-Pérez's alleviation method needs a lot of memory space to store $(2+8 \times n \times \epsilon^{-1})$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2+8 \times n)$ vertices for the *VGraph*. The accuracy is defined by ϵ whose value is very small and n is the number of obstacles in workspace. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time.
- The *Extended VGraph Algorithm* has been presented to solve the drawbacks of the *VGraph Algorithm*.

1.2 Literature Review

The simplest obstacle avoidance algorithm uses the *Generate and Test Method* [16] [28]. A simple path from *Start* to *Goal* is hypothesized and is tested for potential collisions. If a collision is expected, a new path is considered. This process is repeated until no collisions are expected along the new proposed path. In the case of a manipulator, such an algorithm can be described in three steps:

1) Calculate the volume swept out by the manipulator along the proposed path.
2) Determine the overlap between obstacles and the swept volume by a manipulator.
3) Propose a new path.

The first step is self explanatory. The second step is known as an *Interference Detection* [7], detecting the overlap between the obstacles and the swept volume by manipulator. The whole 3 steps are known as the *Swept Volume Method* [28]. Lozano-Pérez [28] and Faverjon [16] have pointed out several difficulties and drawbacks of the *Swept Volume Method*. First, it is quite difficult to model obstacles and a manipulator within resonably short computational time and allowable accuracy. Calculating the volume swept out by a manipulator with revolute joints is a hard job. It can be also difficult to determine whether the swept volume and obstacles overlap. Another fundamental drawback lies in the relationship between the second and the third steps. Each proposed path provides only local information about potential collisions, for example, the shape of the intersections of the volumes involved, or the identity of the obstacle giving rise to the collision. As the manipulator consists of several linked parts, it is difficult to find good heuristics to modify the paths. This lack of a global view can result in an expensive search of the space of possible paths with a very large upper bound on the worst case length of the path. For these reasons, Udupa [59] uses a *Growing Transformation Method* on obstacles to compute approximations to the forbidden regions for the three-dimensional reference point of a three degree of freedom subset of a manipulator [28]. The system maintains a variable resolution description of the legal positions of the reference point. Safe paths for the subset manipulator are found by recursively introducing intermediate goals into a straight line path until the complete path is in free space. This method has two drawbacks pointed by Lozano-Pérez [28] [30].

1) Since the complete manipulator has more than three degrees of freedom, the three-dimensional forbidden regions cannot model all the constraints on the manipulator. When a trajectory fails, Udupa's system makes a correction using manipulator dependent heuristics. The use of heuristics tends to limit the performance of the algorithm in cluttered spaces.
2) The recursive path finder uses only local information to determine a safe path and therefore suffers from some of the same drawbacks as the *Swept Volume Method.*

Lozano-Pérez [28] generalized the ideas of Udupa [59] to the whole manipulator. His algorithm uses a more accurate growing operation to compute the forbidden regions in both two and three-dimensions. It introduces a graph searching technique for path finding, which produces optimum two-dimensional paths when only translations are involved. The algorithm is then generalized to deal with three-dimensional obstacles and extended to deal uniformly with more than three degrees of freedom. However, the generalization to three-dimensions has an unfortunate side effect. The shortest path around a polyhedral obstacle does not in general traverse only vertices of the polyhedron. That is, the shortest path in the *VGraph* whose node set contains only vertices of the grown obstacles is not guaranteed to be the shortest collision-free path. So, Lozano-Pérez [28] proposed a method which is to introduce additional vertices along the edges of the grown obstacles so that no edge is longer than a prespecified maximum length. This approach has some drawbacks. It is difficult to decide how many vertices should be added along the edges of the grown obstacles and the additional vertices need much more computational time for the *VGraph* search.

Brooks [8] solves the *Findpath* problem by good representation of free space; Ahuja [4] and Faverjon [16] use an *Octree* for the obstacle avoidance. Brooks [10] presents an algorithm for polyhedral obstacles and a moving object with two translational and one rotational degrees of freedom. Wong and Fu [63] present a methodology for three-dimensional collision-free path planning by which planning is done in the three-dimensional orthogonal two-dimensional projections of a three-dimensional environment. Peshkin and Sanderson [53] present an algorithm that efficiently finds the externally visible vertices of a polygon and the range of angles. Chung and Saridis [11] present the *Recursive Compensation Algorithm* to solve the drawback of the *VGraph Algorithm*.

2. METHODOLOGY

2.1 The Interference Detection

Boyse [7] presents two types of interference checking: detection of intersections among objects in fixed positions and detection of collisions among objects moving along specified trajectories. The first type of interference checking plays an important role in the *Interference Detection* of the Grown Space Obstacles and the second type of interference checking plays an important role in *Obstacle Avoidance*. To detect a collision between two objects, it is sufficient [7] to detect a collision of an edge on one object with a face of the other or vice-versa. Because a face consists of its interior and a boundary, collision of a face and edge occurs in one of two ways; the edge comes into contact either with the interior of the face or with the boundary of the face. The two cases are shown in Fig. 2.1.1 [7].

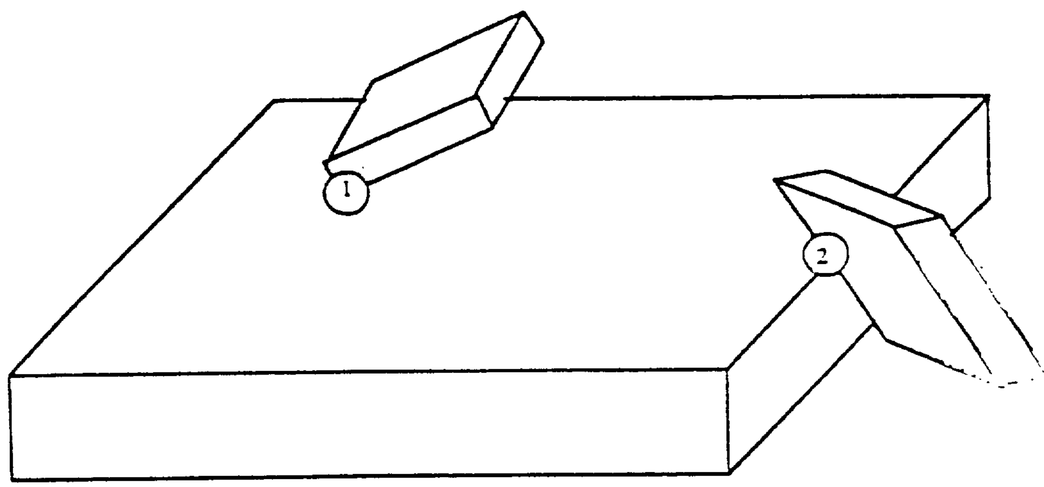

Fig. 2.1.1 Interference Detection

The collision detection algorithm [7] considers each of these two possible situations as follows:

1. *Edge contacts face interior.* Because edges are straight line segments and faces are planar, contact must occur at an endpoint of the edge. Assuming an edge moving relative to a stationary face, collision can be detected by determining the locus of each endpoint of the moving edge and examining these loci (space curves) to see whether either one intersects the face.
2. *Edge contacts face boundary.* Again assume an edge moving relative to a stationary face and note that the locus of this moving edge generates a surface in space. Collision is detected by examining the

boundary of the face to see if it intersects the surface generated by the moving edge.

The first type of interference will be detected by the *Dead Node*, defined as a vertice that is located in the object, shown in Fig. 2.1.2. To be a *Dead Node*, its boundary condition and its boundary equation should be satisfied.

(i) boundary condition

$$x_{min} < P_x < x_{max}$$

$$y_{min} < P_y < y_{max}$$

(ii) boundary equation

$$f_1(x,y) \cdot f_2(x,y) \cdot f_3(x,y) \cdot f_4(x,y) > 0$$

The second type of interference can be detected by checking the *Line Intersection* [57]. The straight forward way [57] to solve this problem is to find the intersection point of the lines defined by the line segments, then check whether this intersection point falls between the end points of both of the segments. In terms of the variables in Sedgewick's algorithm [57], it is easy to check that the quantity $(dx \cdot dy_1 - dy \cdot dx_1)$ is 0 if p_1 is on the line, positive if p_1 is on one side, and negative if it is on the other side. The same holds true for the other point, so the product of the quantities for the two points is positive if and only if the points falls on the same side of the line, negative if and only if the points fall on different sides of the line, and 0 if and only if one or both points fall on the line.

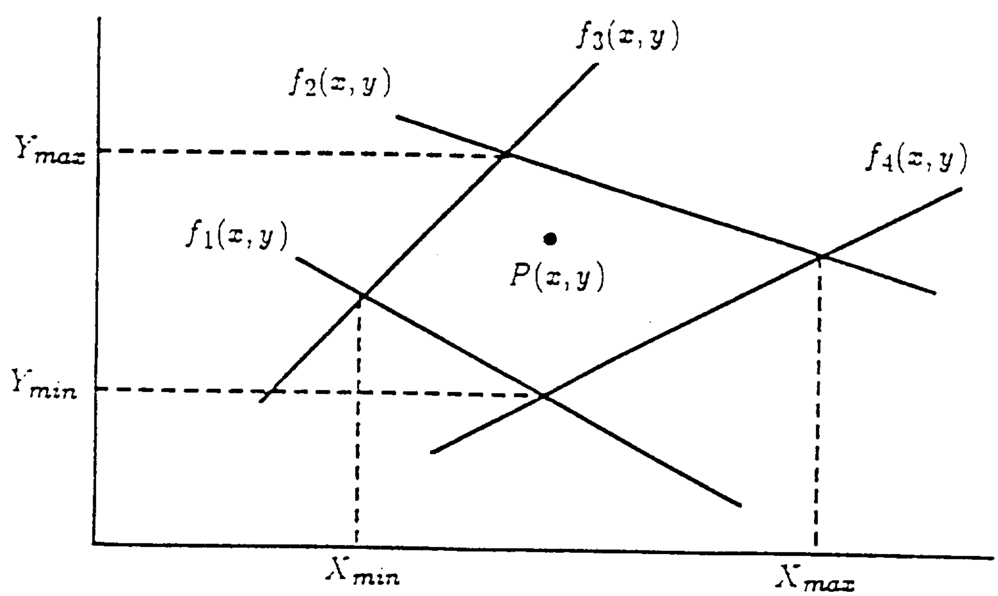

Fig. 2.1.2 Dead Node

2.2 The Grown Space Obstacles

Most of the work in path planning will be done in the field of building the *Configuration Space Obstacles* rather than searching graph. Therefore, it is clear that the representaion of the objects [4] plays a major role in determining the feasibility and performance of any intersection or collision detection method using that representation. Udupa [58] was the first to approach the *Findpath* by explicitly using transformed obstacles and a space where the moving object is shrunked to be a point. Udupa used only rough approximation to the actual *Configuration Space Obstacles* and had no direct method for representing constraints on more than three degrees of freedom [30]. Lozano-Pérez [29] [30] shows that algorithms for computing the *Grown Space Obstacles* in 2D have time complexity $O(v)$, and the algorithms for computing the *Grown Space Obstacles* in 3D have time complexity $O(v^2 \log v)$, where v is the total number of vertices. Considering time complexity, it is much better to find a collision-free path projected in 2D rather than in 3D. Therefore, the *Orthogonal Projection Method* is proposed to build the *Configuration Space Obstacles* in 3D, where three orthogonal cameras are used to build the *Configuration Space Obstacles*. To avoid building the *Grown Space Obstacles* of unnecessary objects in 3D has a lot of advantages. See the section 2.6 for these advantages. A final *Configuration Space Obstacles* in 3D will be reconstructed from the three *Configuration Space Obstacles* in 2D.

Workspace A (Fig. 2.2.1, Fig. 2.2.2 and Fig. 2.2.3) demonstrates Udupa's idea to build the *Grown Space Obstacles* in 2D. Fig. 2.2.1 describes *Workspace A* with three obstacles and the initial and goal states of a moving object. Fig. 2.2.2 describes how to build the *Grown Space Obstacles* with respect to a reference point. The moving object is applied to the boundary of each object and the reference point is traced to obtain the *Grown Space Obstacles*. So, the moving object is shrunked to be a point and the grown geometric objects are obtained, called *Grown Space Obstacles*, that represent all the positions of the moving object that cause collision with the obstacles. Fig. 2.2.3 describes the final *Grown Space Obstacles* for *Workspace A*. The advantage of this formulation [30] is that the intersection of a point relative to a set of objects is easier to deal with than the intersection of objects among themselves. Fig. 2.2.4 describes the data structure for *Workspace A*. Representing the positions of rigid objects requires specifying all their degrees of freedom, both translations and rotations. The configuration [30] of a polyhedron is a set of independent parameters that characterize the position of every point in the object. In following sections, the different initial configuration of a moving object makes the different *Grown Space Obstacles*, which result in different *VGraph*.

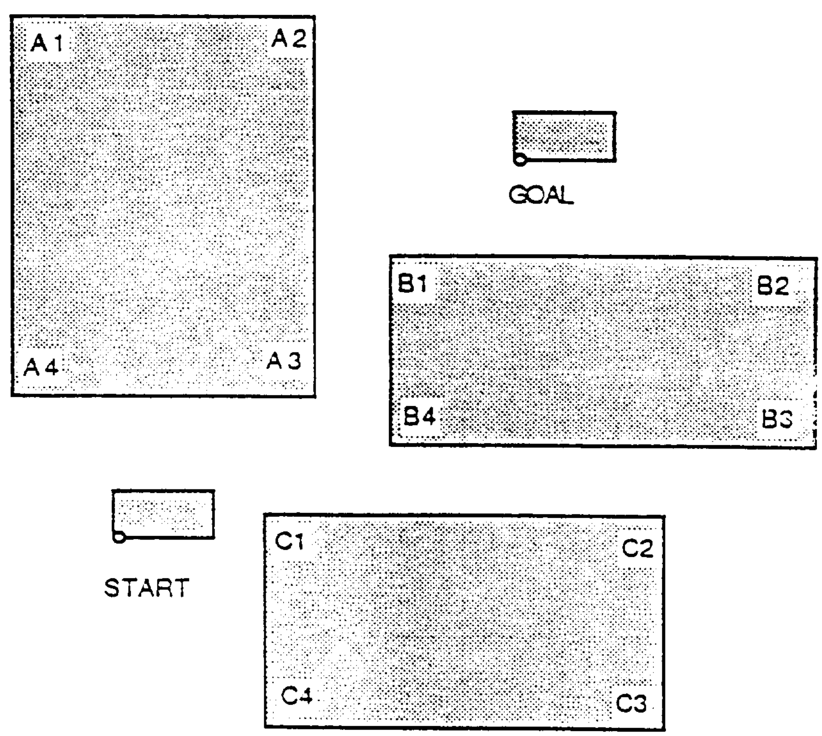

Fig. 2.2.1 A description of Workspace A

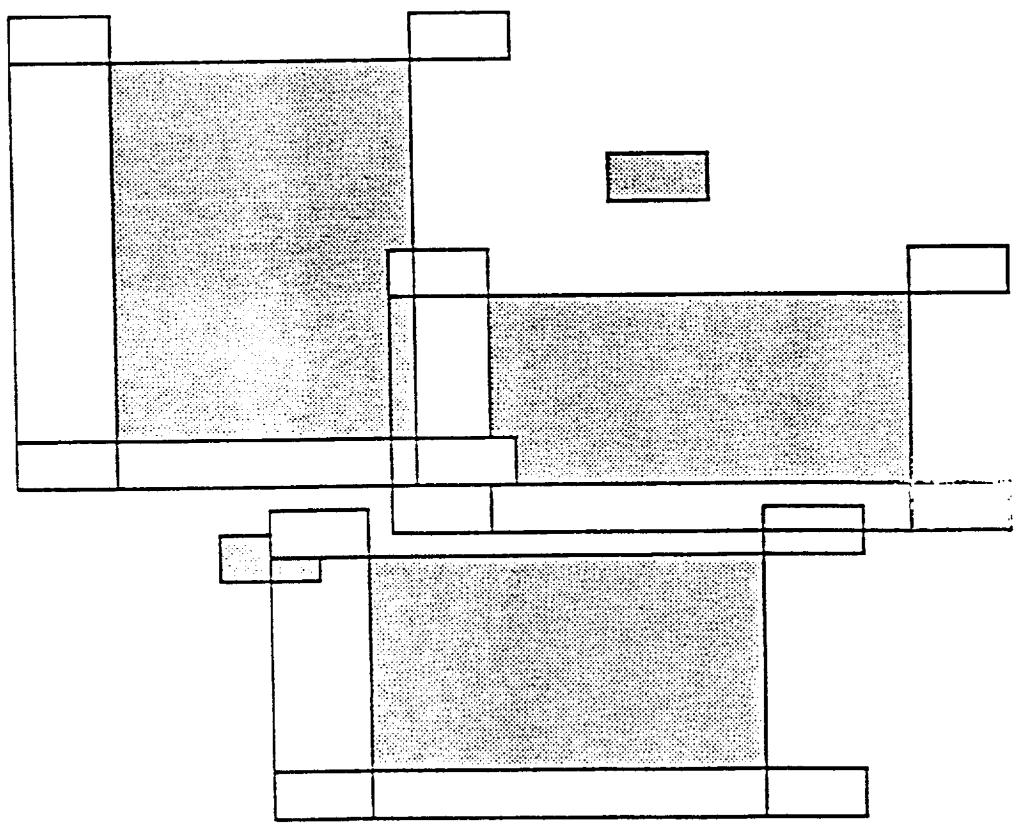

Fig. 2.2.2 A description of Grown Space Obstacles for Workspace A

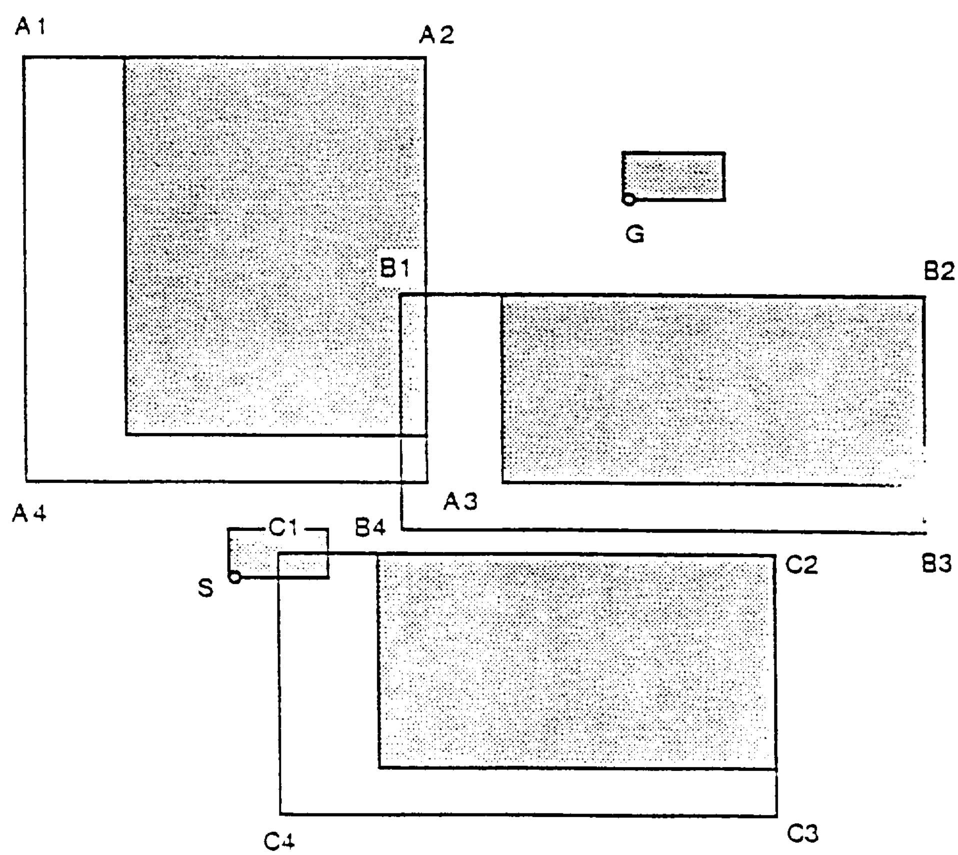

Fig. 2.2.3 A Grown Space Obstacles for Workspace A

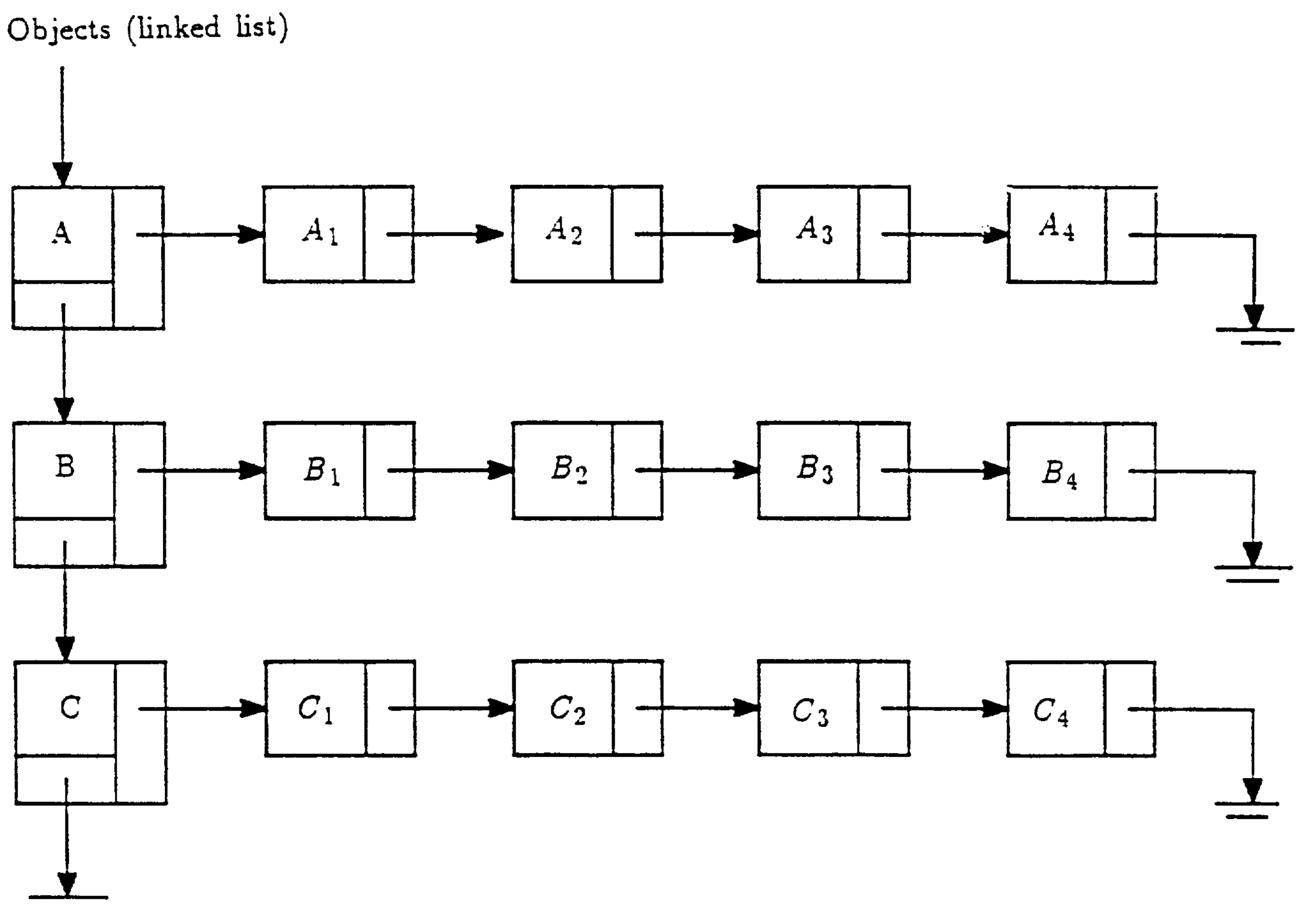

Fig. 2.2.4 Data structure for Workspace A

2.3 The Rotational Grown Space Obstacles

In the previous section, it is known that the different initial configuration of a moving object makes the different *GSpace Obstacles* (*Grown Space Obstacles*), which result in different *VGraphs*. Since the rotation of a moving object can change its initial configuration, each *VGraph* should be built for its rotation of the moving object. Lozano-Pérez [29] [30] presents the θ sliced projection method to build the *GSpace Obstacles* for the rotation of the moving object. Since the moving object can rotate by θ, the number of its *VGraph* ,ξ, can be figured out, where $0 \leq \eta \cdot \theta \leq 2\pi$ and $\eta = 1, 2, \cdots, \xi$. For each η, build its *VGraph* and construct a set of vertices for all vertices for its *VGraph*. Find the visible vertices from this set of vertices by detecting the interferences. This set of visible vertices can build the *VGraph* with the sliced rotation of a moving object.

Let's consider Workspace B, shown in Fig. 2.3.1, having a moving object rotated by θ with respect to Workspace A. Fig. 2.3.2 describes how to build the *Grown Space Obstacles* with respect to a reference point. The moving object is applied to the boundary of each object and the reference point is traced to obtain the rotational *Grown Space Obstacles*. So, the moving object is shrunked to be a point and the grown geometric objects are obtained, representing all the positions of the moving object that cause collision with the obstacles. Fig. 2.3.3 describes the *Grown Space Obstacles* for Workspace B. Workspace A and Workspace B have the same configuration except the initial configuration of the moving object. However, they have the completely different *Grown Space Obstacles*, shown in Fig. 2.2.3 and Fig. 2.3.3. Fig. 2.3.4 describes a data structure of the rotational *GSpace Obstacles*. The vertices of the rotational *GSpace Obstacles*, shown in Fig. 2.3.3, can be obtained from the geometric equations, assuming that h is the horizontal length of a moving object, v is its vertical length, θ is the radian angle between the initial configuration and the rotated configuration with respect to the reference point. And the lower character means the vertices of the obstacles and the upper character means the vertices of the *GSpace Obstacles*, i.e., $a_i = (a_{ix}, a_{iy})$, $A_i = (A_{ix}, A_{iy})$ where $i = 1, \cdots, n$. However, if θ is 0 or $\frac{\pi}{2}$, then the set of A_{odd} equals to the set of A_{even}. Fig. 2.3.4 shows how to design the data structure to store the information on the rotational *Grown Space Obstacles* to save the memory storage.

where $0 \le \theta < \frac{\pi}{2}$

$$
\begin{aligned}
A_1 &= (a_{1x}, a_{1y}) + h \cdot (-cos\theta,\ -sin\theta) \\
A_2 &= (a_{1x}, a_{1y}) \\
A_3 &= (a_{2x}, a_{2y}) \\
A_4 &= (a_{2x}, a_{2y}) + v \cdot (sin\theta,\ -cos\theta) \\
A_5 &= (a_{3x}, a_{3y}) + v \cdot (sin\theta,\ -cos\theta) \\
A_6 &= (A_{5x}, A_{5y}) + h \cdot (-cos\theta,\ -sin\theta) \\
A_7 &= (A_{8x}, A_{8y}) + v \cdot (sin\theta,\ -cos\theta) \\
A_8 &= (a_{4x}, a_{4y}) + h \cdot (-cos\theta,\ -sin\theta).
\end{aligned}
$$

where $\frac{\pi}{2} \le \theta < \pi$

$$
\begin{aligned}
\theta &\longleftarrow \theta - \frac{\pi}{2} \\
h &\longleftarrow v \\
v &\longleftarrow h.
\end{aligned}
$$

Lozano-Pérez points out two important properties of sliced projection:

i) a solution to a Findspace problem in any in the slices is a solution to the original problem, but since the slices are an approximation to the *Grown Space Obstacles*, the converse is not necessarily true;

ii) the slice projection of a *Grown Space Obstacles* can be computed by using the swept volume operation, without having to compute the high-dimensional *Grown Space Obstacles*.

When rotations of a moving object are allowed, the slice projection operation can be used to extend the *VGraph Algorithm* to find safe paths [29].

[Problem Statement 2.3] Assuming that the horizontal length of the moving object is 2, its vertical length is 1 and θ is $\frac{\pi}{6}$ and obstacles are given as in Fig. 2.3.5, draw the rotational *GSpace Obstacles*.

Fig. 2.3.6 - Fig. 2.3.11 draw the rotational *GSpace Obstacles*. The programming list for the simulation of the rotational *GSpace Obstacles* is available in Appendix C and Appendix D.

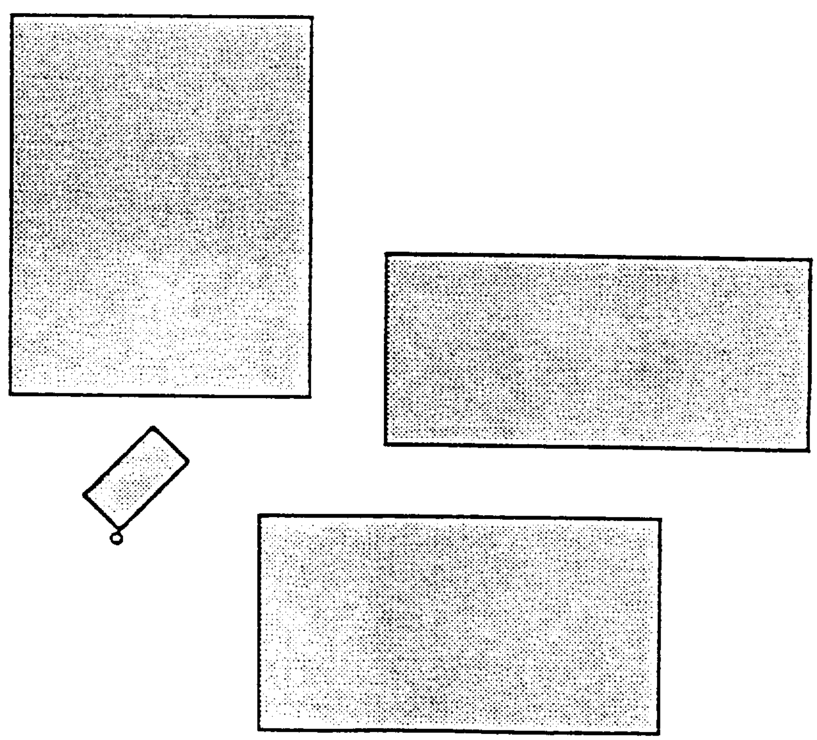

Fig. 2.3.1 A description of Workspace B

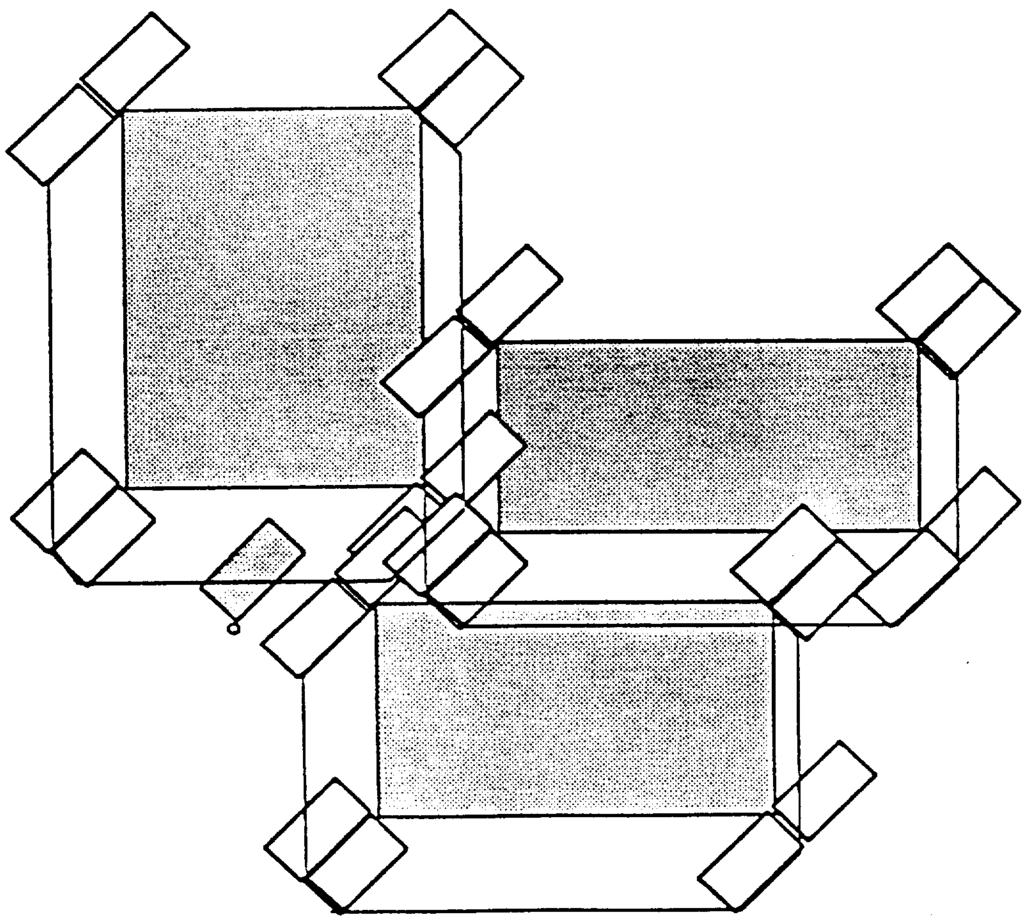

Fig. 2.3.2 A description of the rotational GSpace

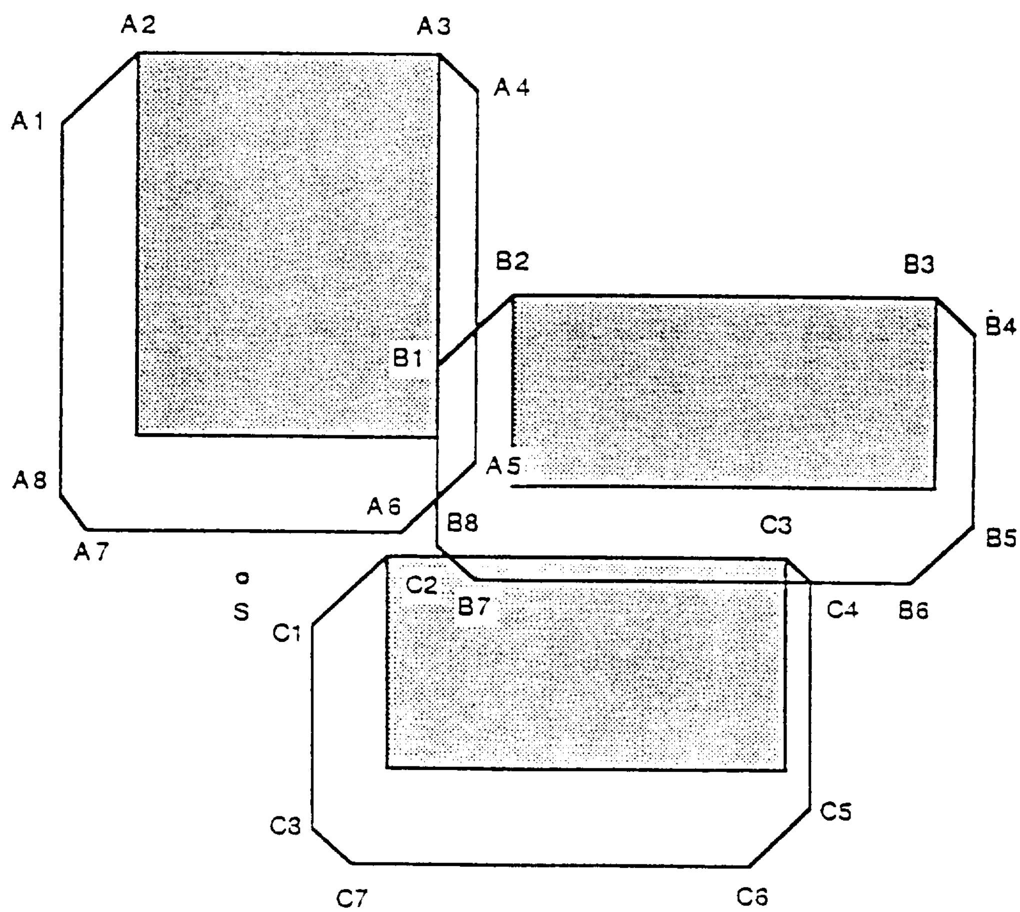

Fig. 2.3.3 A rotational GSpace

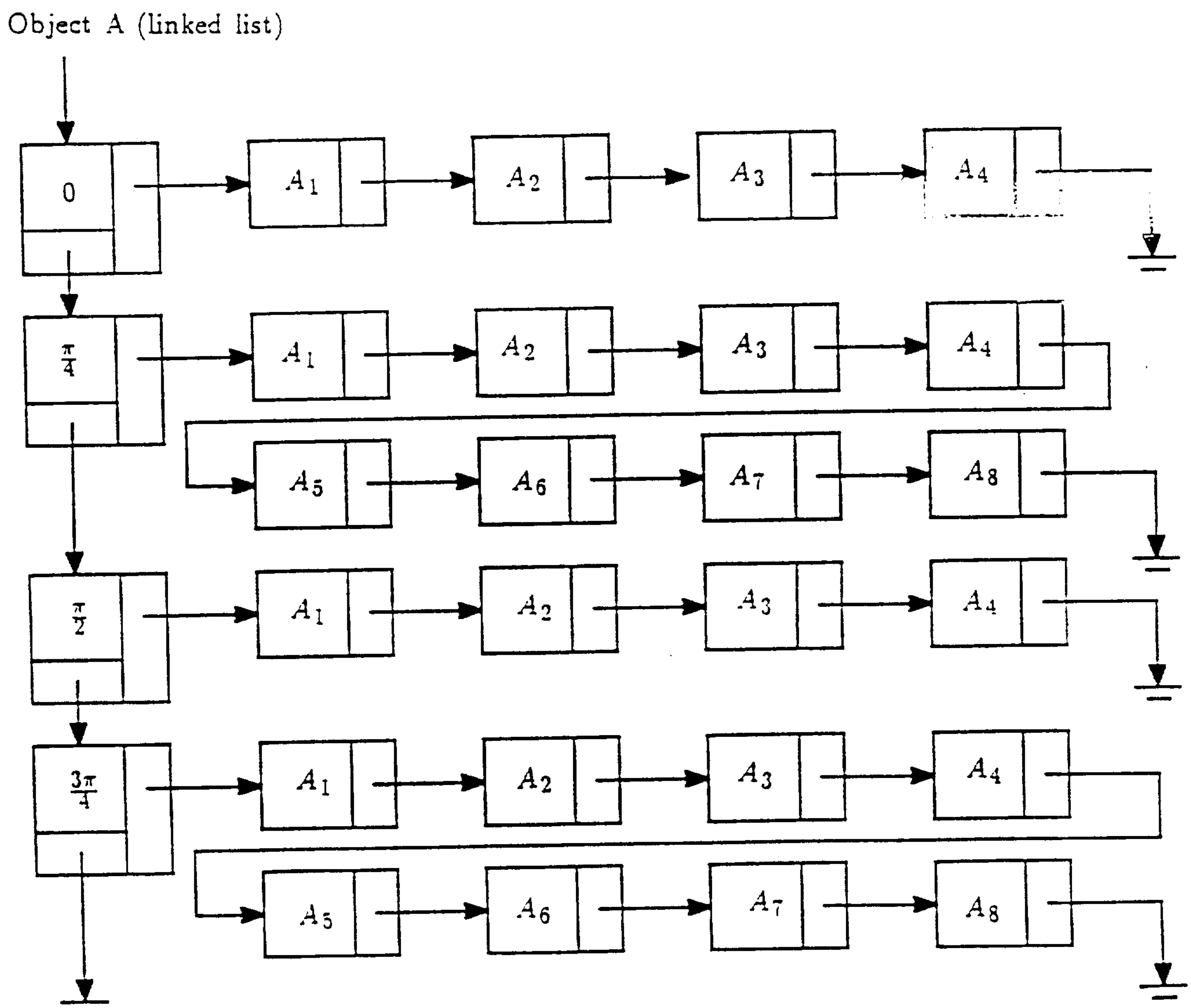

Fig. 2.3.4 Data structure of the rotational GSpace

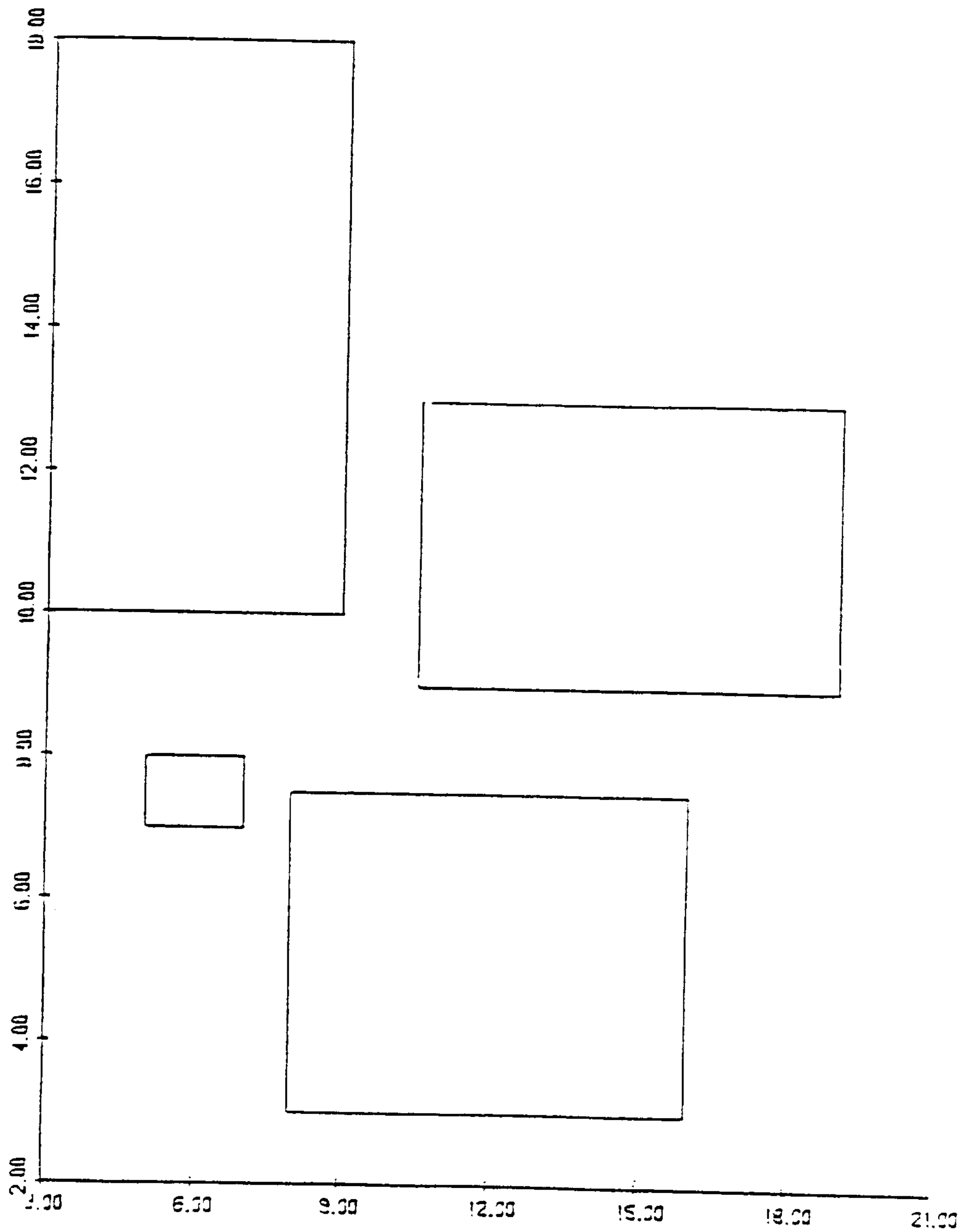

Fig. 2.3.5 Workspace C for the Problem Statement 2.3

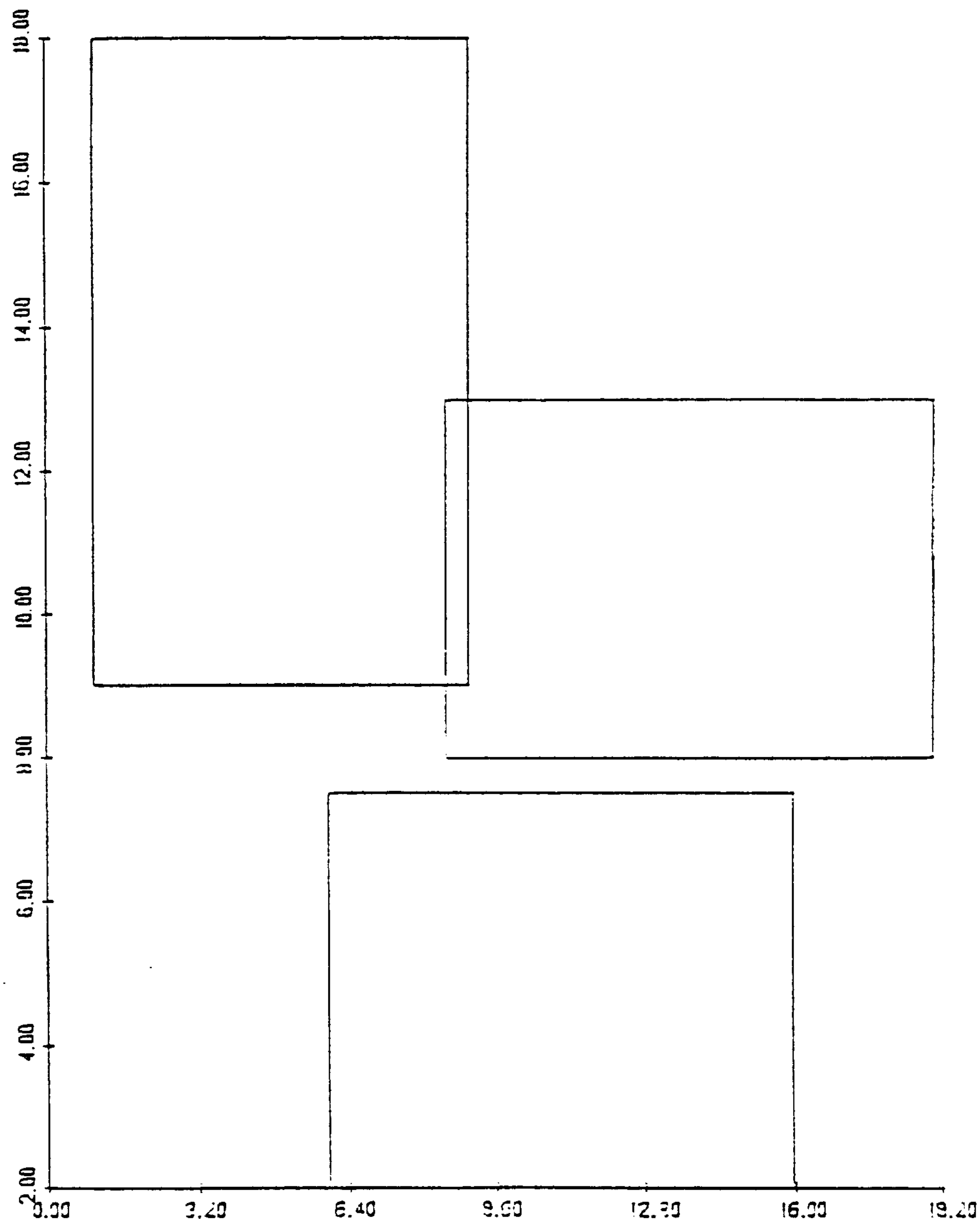

Fig. 2.3.6 A rotational GSpace with 0 sliced

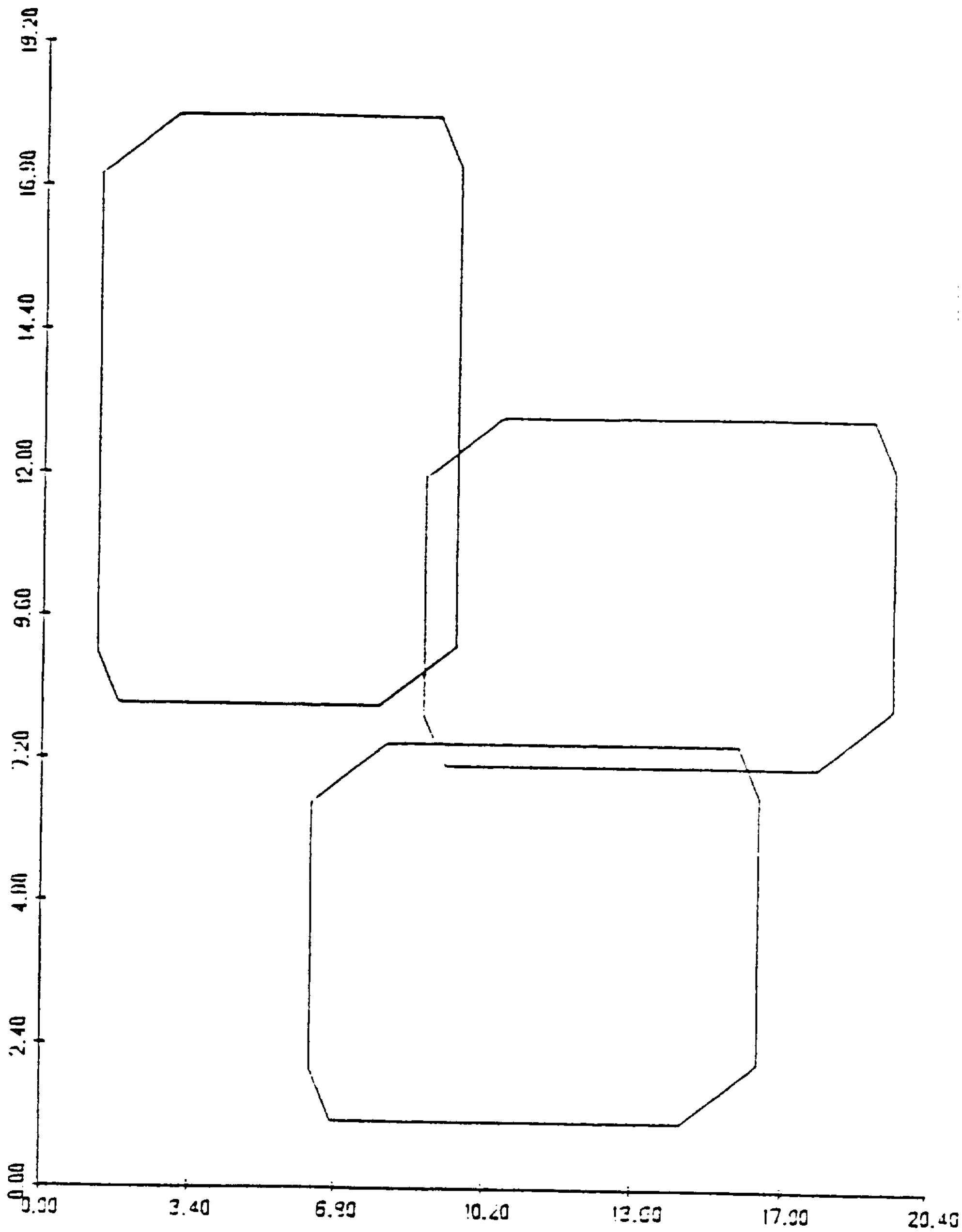

Fig. 2.3.7 A rotational GSpace with $\frac{\pi}{6}$ sliced

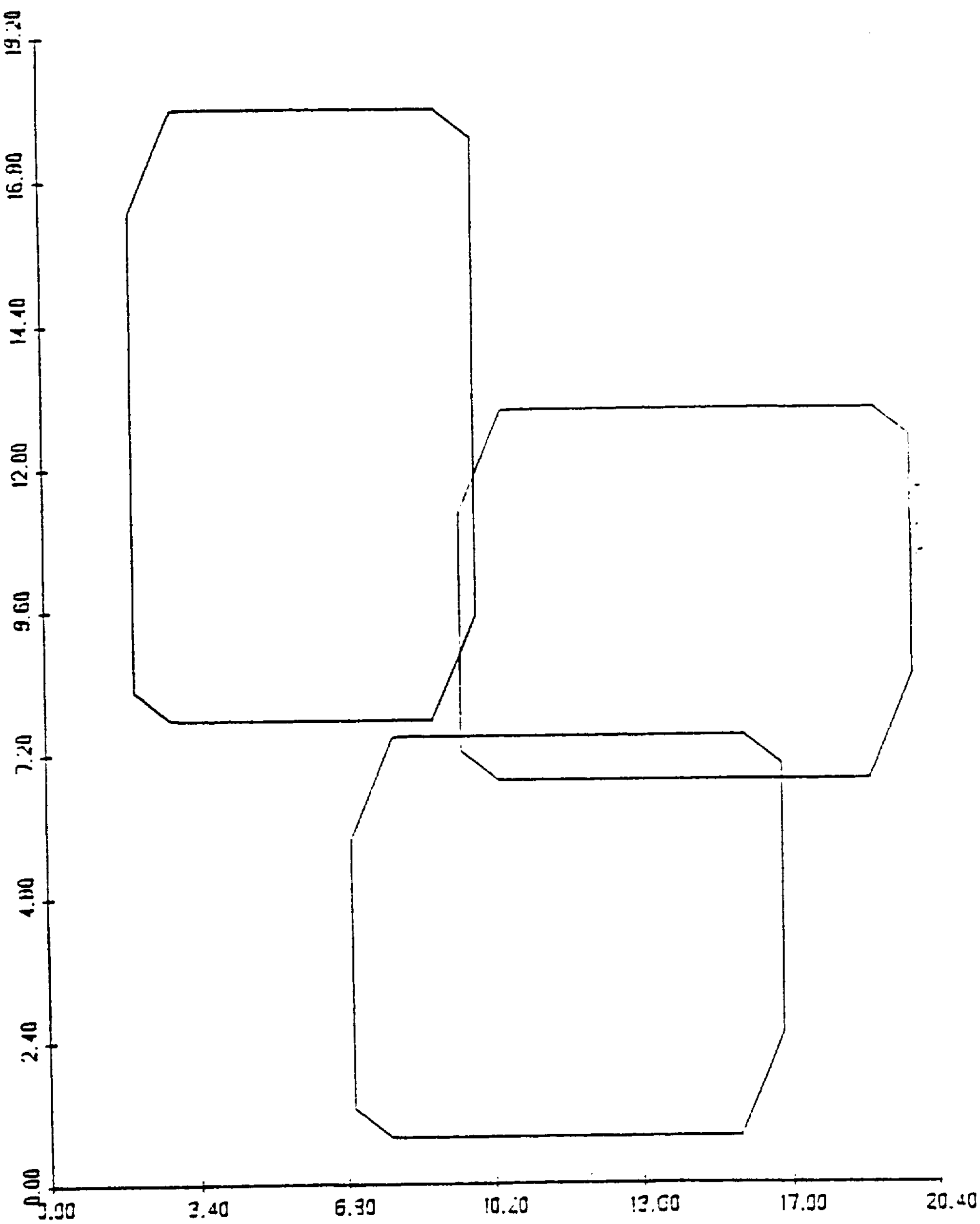

Fig. 2.3.8 A rotational GSpace with $\frac{\pi}{3}$ sliced

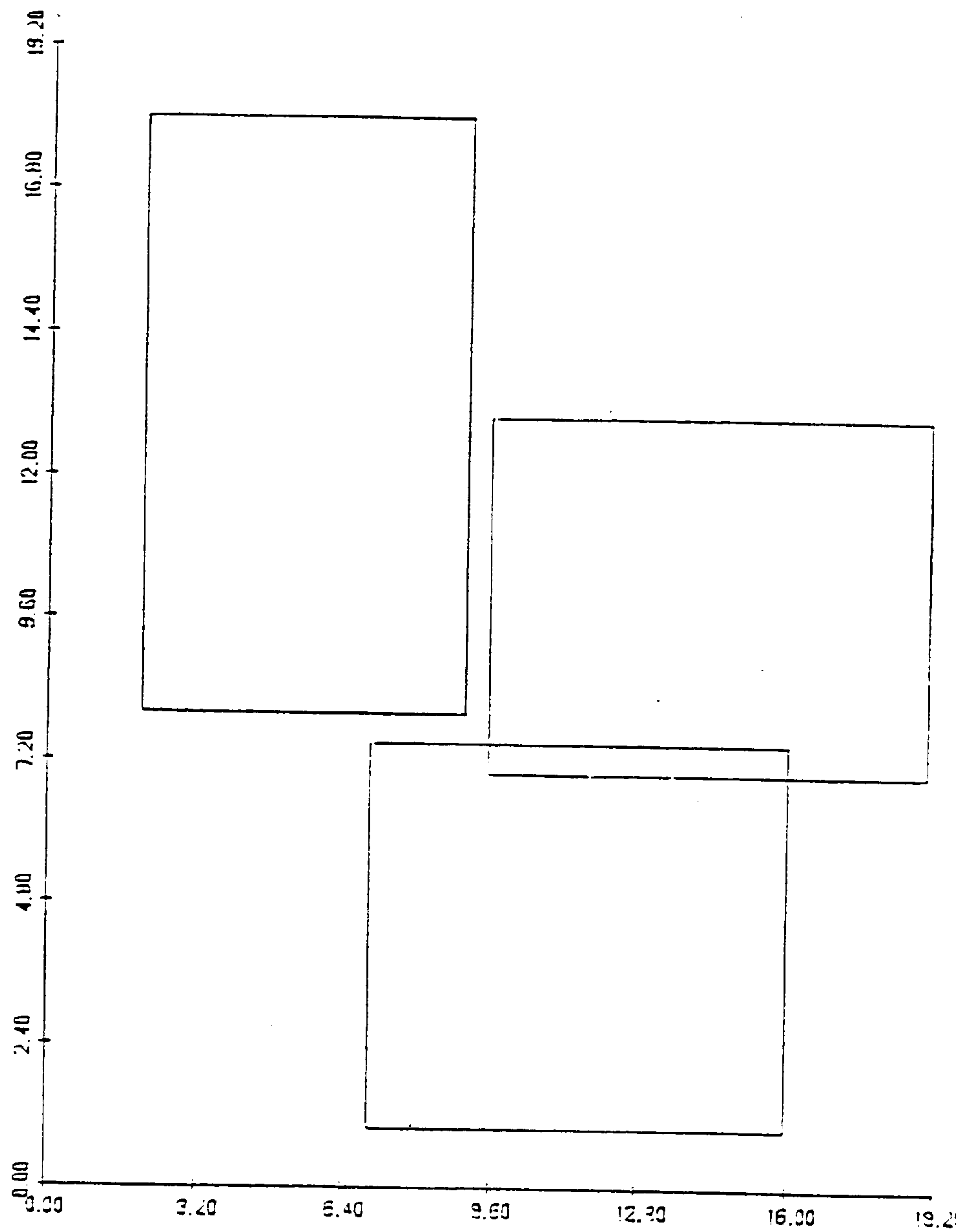

Fig. 2.3.9 A rotational GSpace with $\frac{\pi}{2}$ sliced

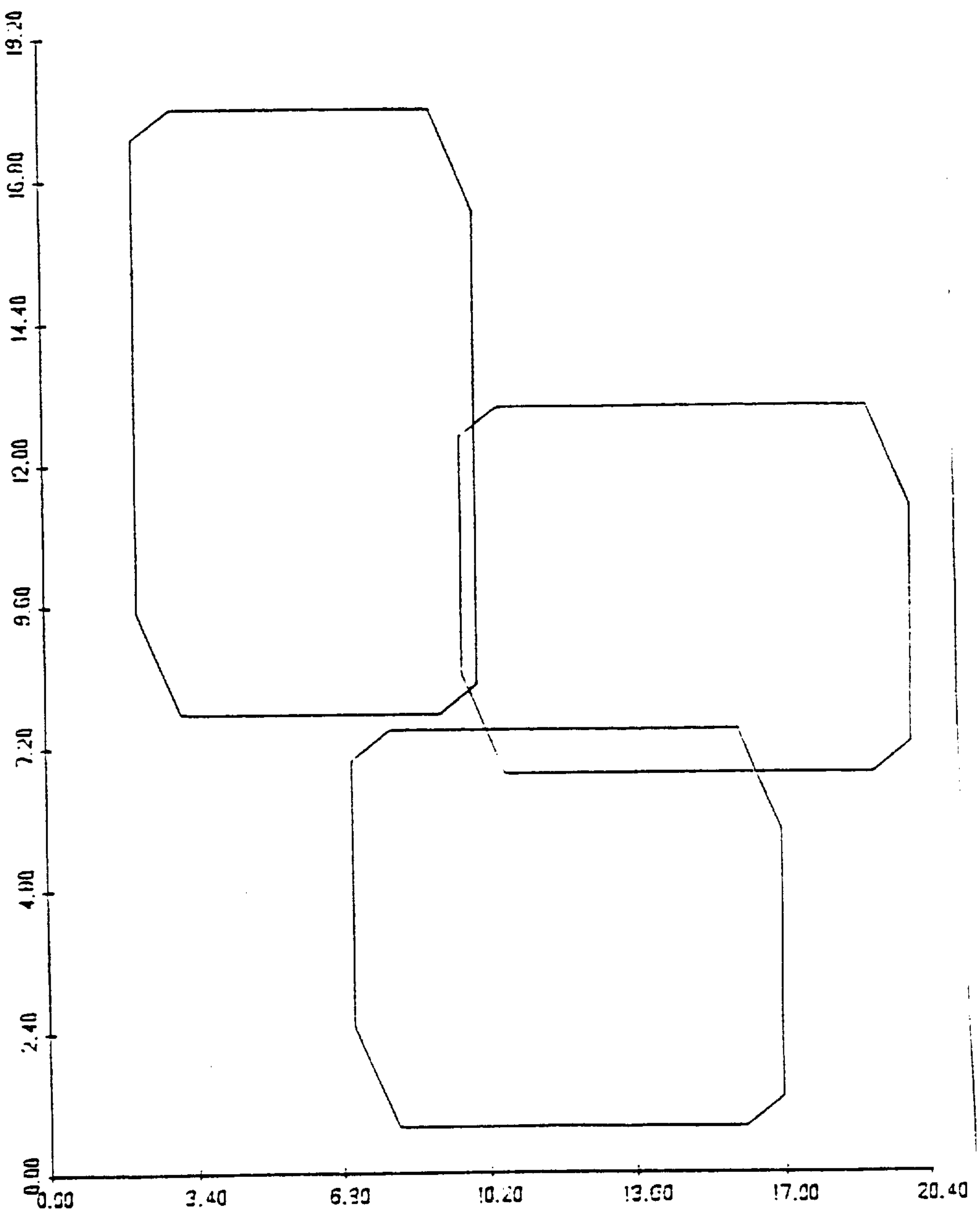

Fig. 2.3.10 A rotational GSpace with $\frac{2\pi}{3}$ sliced

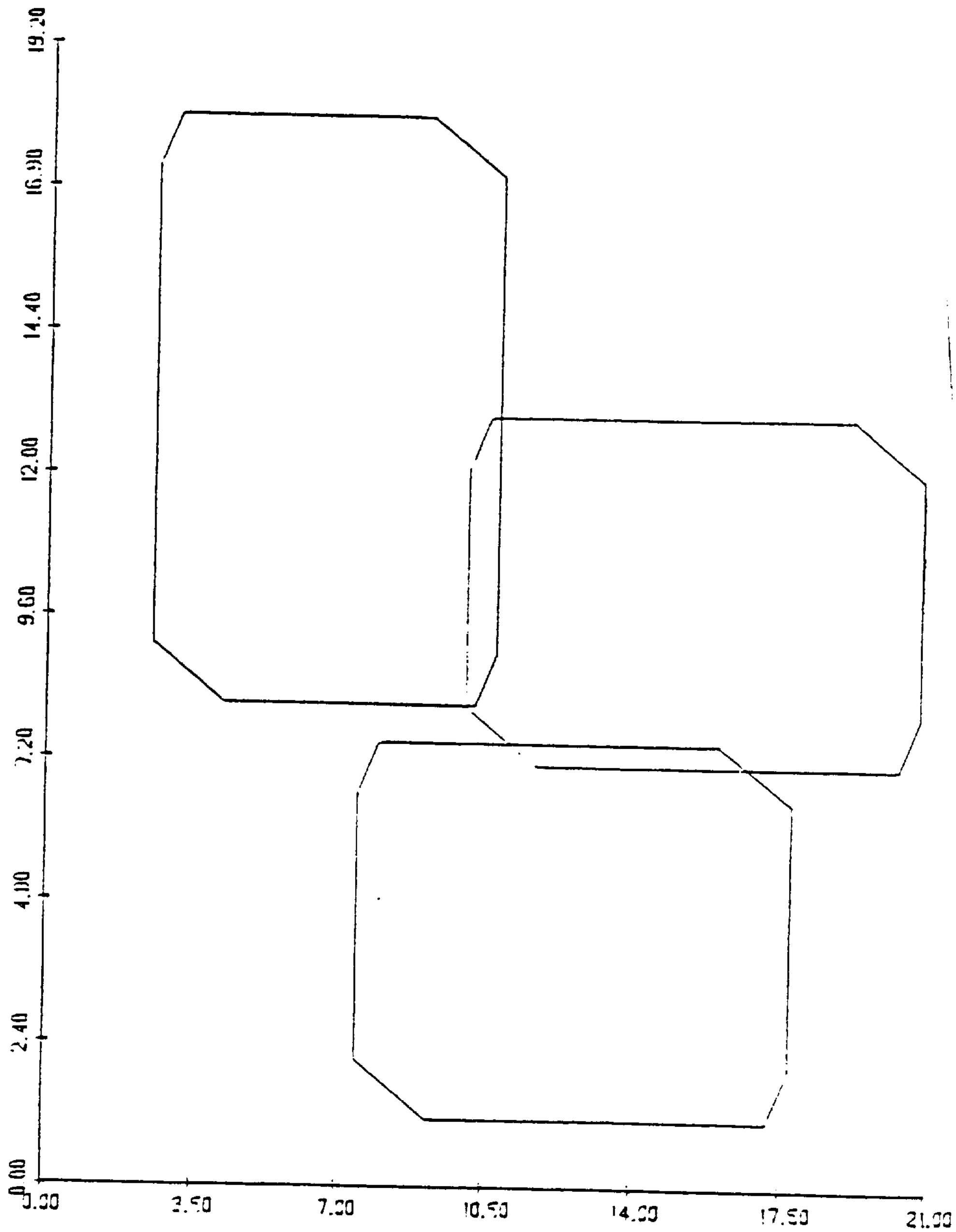

Fig. 2.3.11 A rotational GSpace with $\frac{5\pi}{6}$ sliced

2.4 The VGraph Algorithm

Consider the problem for a moving object to find the collision-free shortest path from the *Start* to the *Goal*. It is desirable that a method represent the *Grown Space Obstacles* with a graph in order to find the shortest path from *Start* and *Goal*. The important property of this path is that it is composed of a straight line joining the *start* point and the *goal* point via a possibly empty sequence of vertices of obstacles [28]. The undirected graph [28] is denoted by $VG(N, L)$ where N is the union of S, G and all the obstacle's vertices. The link set, L, is the set of all the links (N_i, N_j) such that a straight line exits connecting the $i^{\underline{th}}$ element of N to the $j^{\underline{th}}$ without overlapping any obstacles. The graph $VG(N, L)$ is thus called the *visibility graph* (*VGraph*) of N, since the connected vertices in the graph can see each other. The *VGraph* for Workspace A is shown in Fig. 2.4.1. To build the *VGraph*, first construct the *Grown Space Obstacles* and then detect the interference of its vertices. If not interfered, the vertice is sent to the set of versible vertices. The *VGraph* can be built by this set of the visible vertices.

The *VGraph Algorithm* requires that the moving object be a point while the obstacles are the forbidden regions for the position of that point [28]. If the moving object is not a point, a new set of obstacles must be computed which are the forbidden regions of some reference point on the moving object. These new obstacles must describe the locus of positions of this reference point which would cause a collision with any set of the original obstacles. The method to build the *Grown Space Obstacles* was described in the previous section. This *VGraph Algorithm* could be applied to find the collision-free shortest path in 2D.

The VGraph Algorithm

1. Build the *Grown Space Obstacles.*
2. Find the visible vertices of the *Grown Space Obstacles.*
3. Build the *VGraph* with the visible vertices.
4. Search the *VGraph* by the graph search algorithm.

It is known that the different initial configuration of a moving object makes the different *GSpace Obstacles*, which result in the different *VGraph*. Since the rotation of a moving object can change its initial configuration, each *VGraph* should be built for its rotation of the moving object. Consider the θ sliced rotation of the moving object. Since the moving object can rotate by θ, the number of its *VGraph* , ξ, can be figured out, where $0 \leq \eta \cdot \theta \leq 2\pi$ and $\eta = 1, 2, \cdots, \xi$. For each η, build its *VGraph* and construct a set of vertices for all vertices for its *VGraph*. Find the visible vertices from this set of vertices by detecting the interferences. This set of visible vertices can build the *VGraph* with the sliced rotation of a moving object. The algorithm is the following:

```
Procedure BuildVGraph(var VGraph, List)
Comments
•  List is a linked list for the set of visible vertices.
•  VGraph is (n×n) array to store the cost between two visible vertices
   and ∞ means that two vertices are invisible.
Begin
    From ← List
    To ← List
    VGraph ← ∞
    while From ≠ nil do {
        if not DeadNode(List,From↑.Node)
           then {
                To ← List
                while To ≠ nil do {
                    if (From = To) or
                       DeadNode(List,From↑.Node) or
                       DeadNode(List,To↑.Node) or
                       Interference(List,From↑.Node,To↑.Node)
                       then { do nothing. }
                       else VGraph ← cost between two visible
                                     vertices
                    To ← To↑.Next } }
        From ← From↑.Next }
End
```

The shortest path from the *start* to the *goal* in this *VGraph* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* in section 2.7 is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

[Problem Statement 2.4] Consider the problem, shown in Fig. 2.2.1, assuming that the objects are polyhedrons and their visual informations are available and they are represented by vertices. Find the collision-free shortest distance from *Start* to *Goal* with $\frac{\pi}{4}$ sliced rotation.

The programming list of this simulation is available in the Appendix A. The following result for the Problem Statement 2.4 comes from the file [PATH] in Appendix B.

Table 2.4.1 Simulation result of the VGraph algorithm.

The shortest path is calculated by the VGraph Algorithm.			
Start Node = 1, Goal Node = 27,			
Path represented by internal nodes: $1 \longrightarrow 16 \longrightarrow 6 \longrightarrow 27$			
From	To	Cost	Rotation
Start in 0 sliced	A_3 in $\frac{\pi}{2}$ sliced	4.125	$\frac{\pi}{2}$
A_3 in $\frac{\pi}{2}$ sliced	B_1 in $\frac{\pi}{2}$ sliced	5.025	0
B_1 in $\frac{\pi}{2}$ sliced	Goal in 0 sliced	4.031	$-\frac{\pi}{2}$
The total cost between Start and Goal = 13.179			

Fig. 2.4.2 shows the collision-free shortest path with $\frac{\pi}{4}$ sliced rotation by the *VGraph Algorithm*. The path with $\frac{\pi}{4}$ sliced rotation has 13.179 *Euclidean* distance, while the path without sliced rotation has 25.452 *Euclidean* distance. The path segment with sliced rotation is described in the Table 2.4.1, the path segment without sliced rotation is $\{Start \rightarrow C_1 \rightarrow B_3 \rightarrow B_2 \rightarrow Goal\}$. Hence, the sliced rotation of the moving object can shorten the *Euclidean* distance. However, there is a trade off between accuracy and speed. If the small sliced rotation is considered, then the better approximation to the shortest path can be obtained, but more memory space to store each *VGraph* is needed.

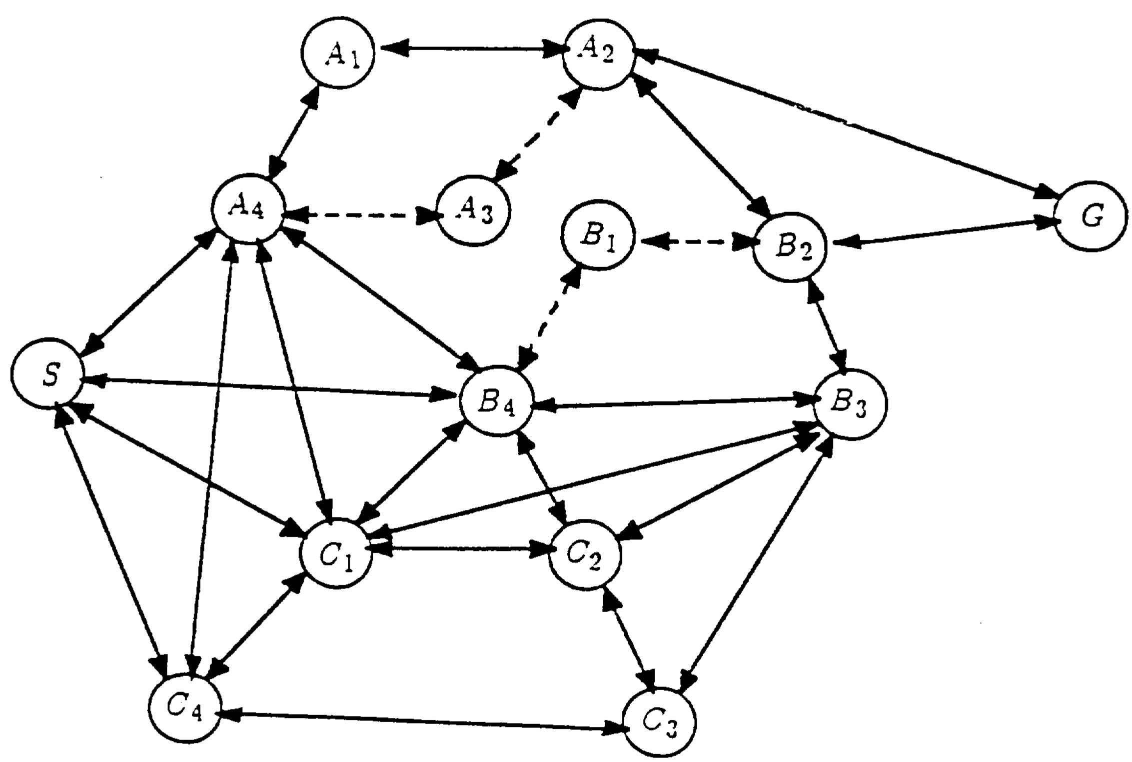

Fig. 2.4.1 A VGraph for Workspace A

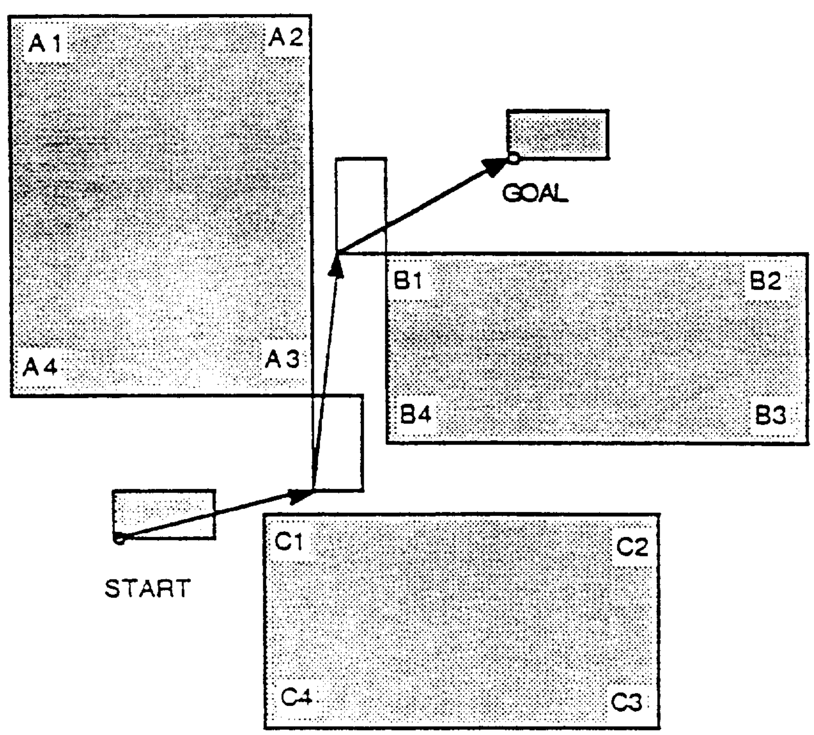

Fig. 2.4.2 The collision-free shortest path for Workspace A

2.5 The Graph Search Algorithm

There are a number of control strategies for finding a path through a graph. The fundamental control problem is to select an appropriate database and the applicable rules to apply in the search for a satisfactory path. The path finding problem has usually been approached in one of two ways [22], the *Mathematical Approach* and the *Heuristic Approach*. The *Mathematical Approach* deals with properties of abstract graphs and with algorithms that follow an orderly examination of the nodes of a graph to find the minimum cost path. The *Heuristic Approach*, on the other hand, typically uses special knowledge about the problem. The efficiency with which a path is found increases as the knowledge becomes closer to being complete. An important point to note is that the *Heuristic Approach* generally is not able to guarantee that the minimum cost path will be found.

General Graph Search Algorithm [61]

1. Put the *START* node on *OPEN*.
2. If *OPEN* is empty, exit with failure.
3. Select a node n from *OPEN* and put it on *CLOSE*.
4. Expand n and put some of its successors in *OPEN* with a pointer back to n. If any of these successors is a *GOAL* node, exit with the solution by tracing back its pointers.
5. Go to step 2.

Pearl [52] describes the main features of the different graph search algorithms as *Hill-climbing, Depth-first, Backtracking, Backmarking, Breadth-first, Uniform-cost* and *Best-first*. While sharing this common framework, the algorithms differ at least in one of the following points:

- The number of successors generated
- The node from *OPEN* selected for expansion
- The particular management strategies used for cleaning up *CLOSED*

The following remarks [52] should be considered to select a search algorithm for path planning purpose. First, optimality is a concept which seems to be opposed to time and storage efficiency. The only algorithm in which these concepts are somehow compatible is the *A* Algorithm*, as long as a good enough heuristic evaluation function is used. Second, the algorithms in which the goodness of the solution can be established are the *breath-first* and the *uniform-cost* ones. In most cases this has a high computational cost. Third, to reduce the time and storage requirements of the search algorithms, it is necessary to make them use information which is beyond the pure graph structure. *Hill-climbing*, *backmarking* and *best-first* are the search algorithms which can incorporate information about the particular domain. There are, so, three facts which strongly recommend *best-first* algorithms as the most appropriate ones for path planning purposes: their adequacy for incorporating information belonging to the particular domain being dealt with, their capability of converging to either optimal or non-optimal solutions and their time and storage efficiency. These are the main reasons [61] that led us to select *best-first* algorithms as the most appropriate for path planning. The *A* Algorithm* is probably the most widely used best-first graph search procedure. The reason for its success [61] lies in its *simplicity*, its *generality*, the *optimality* of its solutions, and the fact that if its h function is *admissible* (that is, it never overestimates the cost of a subpath), the *A* Algorithm* is *optimal* [47]. For example, the criterion used in the *VGraph* can taken into account the distance to be travelled in the *Configuration Space* but also the costs assigned to the change of speed [16].

The total estimate $\widehat{f}(n)$ is an estimate of the cost of a minimal cost path from s to g node constrained to go through node n, and can be expressed as $\widehat{f}(n) = \widehat{g}(n) + \widehat{h}(n)$, where $\widehat{g}(n)$ is an estimate of the cost $g(n)$ of a minimcal cost path from s to n and $\widehat{h}(n)$ is an estimate of the cost $h(n)$ of a minimal cost path from n to a goal node. $\widehat{g}(n)$ is constructed step by step by the algorithm, whereas $\widehat{h}(n)$ is obtained from the heuristic information.

A* Algorithm [41]

1. Put the start node s on a list called *OPEN*. Set $\widehat{g}(s) \leftarrow 0$ and $\widehat{f}(s) \leftarrow \widehat{h}(s)$.
2. If *OPEN* is empty exit with failure; othewise continue.
3. Remove from *OPEN* that node n whose $\widehat{f}$ value is smallest and put it on a list called *CLOSED*. (Resolve ties for minimal $\widehat{f}$ values arbitrarily, but always in favor of any goal node.)
4. If n is a goal node, exit with the solution path obtained by tracing back through the pointers; otherwise continue.
5. Expand node n, generating all of its successors. (If there are no successors, go to Step 2.) For each successor n_i, compute $g_i \leftarrow \widehat{g}(n) + c(n, n_i)$.
6. If a successor n_i is not already on either *OPEN* or *CLOSED*, set $\widehat{g}(n_i) \leftarrow g_i$ and $\widehat{f}(n_i) \leftarrow g_i + \widehat{h}(n_i)$. Put n_i on *OPEN* and direct a pointer from it back to n.
7. If a successor n_i is already on *OPEN* or *CLOSED* and if $\widehat{g}(n_i) > g_i$, then update it by setting $\widehat{g}(n_i) \leftarrow g_i$ and $\widehat{f}(n_i) \leftarrow g_i + \widehat{h}(n_i)$. Put

n_i on *OPEN* if it was on *CLOSED* and redirect to n the pointer from n_i.

8. Go to Step 2.

It is possible to prove [47] that if, for every node n, $\widehat{h}(n)$ is a lower bound on the cost $h(n)$ of a minimal cost path from node n to a goal node, then the A^* *Algorithm*, is *admissible*, i.e. it always finds an optimal path. Moreover, it is possible to simplify the A^* *Algorithm* by making a further assumption on the estimate $\widehat{h}$: for any two nodes m and n which are connected by an $arc(m, n)$ we have $\widehat{h}(m) - \widehat{h}(n) \leq c(m, n)$. This assumption is called the *consistency assumption* and its meaning is that, by moving from a node to any successor, we must always have a better estimate. Therefore the A^* *Algorithm* with the consistency assumption expands fewer than N nodes and, hence, it runs in $O(N)$ steps [41].

Theorem 2.4.1 [41] For all N there exists a search graph G_N of size N, with positive costs and estimates which are lower bounds ($\widehat{h}(n) \leq h(n)$ for each n), on which the A^* *Algorithm* runs for $O(2^N)$ steps.

Martelli [41] presents the *B Algorithm* to modify the A^* *Algorithm* in order to improve its behaviour with nonconsistent estimate. The *B Algorithm* is thus a simple variant of the A^* *Algorithm* and can be obtained from it by substituting steps (1) and (3) with the following steps:

1'. Put the start node s on a list called *OPEN*. Set $\widehat{g}(s) \leftarrow 0$, $\widehat{f}(s) \leftarrow \widehat{h}(s)$, $F \leftarrow 0$.

3'. If there are some nodes in *OPEN* with $\widehat{f} < F$, select among them the node n whose $\widehat{g}$ value is smallest; otherwise, select the node n in *OPEN* whose $\widehat{f}$ value is smallest and set $F \leftarrow \widehat{f}(n)$. (Resolve ties arbitrarily, but always in favor of any goal node.) Remove n from *OPEN* and put it on a list called *CLOSED*.

Theorem 2.4.2 [41] Given any search graph G of size it N, with positive costs and estimates which are lower bounds on the minimal cost ($\widehat{h}(n) \leq h(n)$ for each n), then the *B Algorithm* runs on it for at most $O(N^2)$ steps.

Theorem 2.4.3 [41] Let G be any search graph with positive costs and estimates which are lower bounds on the minimal cost ($\widehat{h}(n) \leq h(n)$ for eah node n). Then, if the A^* *Algorithm* and the *B Algorithm* resolve ties in the same way, the *B Algorithm* does not expand more nodes than the A^* *Algorithm*.

Theorem 2.4.3 assures us that the *B Algorithm* can always be used in place of the A* Algorithm without having any loss in efficiency. In particular, when searching trees or graphs with a consistent estimate, both algorithms will have the same behaviour, but, with a nonconsistent estimate, the *B Algorithm* will, in general, have a much better behaviour than the *A* Algorithm*.

2.6 The Orthogonal Projection Method

Most of the work in path planning will be done in the field of building the *Configuration Space Obstacles* rather than searching graph. Therefore, it is clear that the representaion of the objects [4] plays a major role in determining the feasibility and performance of any intersection or collision detection method using that representation. Lozano-Pérez [29] [30] shows that algorithms for computing the *Grown Space Obstacles* in 2D have time complexity $O(v)$, and the algorithms for computing the *Grown Space Obstacles* in 3D have time complexity $O(v^2 \log v)$, where v is the total number of vertices. Considering time complexity, it is much better to find a collision-free path projected in 2D rather than in 3D.

Some classes [3] [63] of three-dimensional objects, which are of rigid solids of uniform thickness, can be described by the line drawings of three orthogonal projections onto the two-dimensional planes. For these classes of objects, the three-dimensional model can be reconstructed from the three-dimensional objects. However, for other classes of three-dimensional objects [63], the exact reconstruction from line drawings of three orthogonal projections cannot be performed. However, a maximal volume that encloses the volume of the object could be reconstructed. If a three-dimensional point does not collide with the maximal volume, it would not collide with the true object. This leads to a sufficient condition [63] for collision checking, as stated in the following Lemmas.

Lemma 2.6.1 [63]: If the projection of a three-dimensional point is outside the area of the two-dimensional projection of a three-dimensional object in one or more of the three orthogonal subspaces, the three-dimensional point is guaranteed to be outside the volume of the three-dimensional object in the three-dimensional space.

Lemma 2.6.2 [63]: If the projection of some three-dimensional path for the reference point of a three-dimensional moving object is collision free in one or more of the orthogonal projected spaces, then the three-dimensional moving object is collision-free along the three-dimensional path in the three-dimensional space.

Since the unnecessary obstacles, for the Findpath problem in 3D, can be avoided by Lemma 2.6.1 and Lemma 2.6.1, the *Orthogonal Projection Method* can simplify the *VGraph*. So, the *Orthogonal Projection Method* has some advantages. It can shorten the graph search time as well as it can save the memory space to store the *Grown Space Obstacles* and *VGraph*. And advantage is related with its representation. The *Grown Space Obstacles* in 3D can be represented and built by three *Grown Space Obstacles* in 2D. Three processors are assigned for this job and they work so simultaneously that the parallel processing can be expected. Therefore, the *Orthogonal Projection Method* can save more time in building the *Grown Space Obstacles* than any other algorithms that work sequentially. The *Orthogonal Projection Method* has the following steps in turn:

1. Project Objects in 3D onto the projection spaces.
2. Build the *Grown Space Obstacles* in 2D.
3. Select the necessary *Grown Space Obstacles* by Lemma 2.6.2.
4. Reconstruct the *Grown Space Obstacles* in 3D.

However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

[Problem Statement 2.6] Build the *Grown Space Obstacles* in 3D by using the *Orthogonal Projection Method*, assuming that the object in 3D is a polyhedron, shown in Fig. 2.6.1. The object has the following vertices; $P_1(x_1, y_1, z_1)$, $P_2(x_1, y_2, z_1)$, $P_3(x_2, y_2, z_1)$, $P_4(x_2, y_1, z_1)$, $P_5(x_1, y_1, z_2)$, $P_6(x_1, y_2, z_2)$, $P_7(x_2, y_2, z_2)$, $P_8(x_2, y_1, z_2)$, where $x_1 = 7, y_1 = 5, z_1 = 3, x_2 = 14, y_2 = 10, z_2 = 12$.

Fig. 2.6.2 describes three Orthogonal Projections of Workspace D. Fig. 2.6.3 describes the *Grown Space Obstacles* in 2D. Fig. 2.6.4 describes the reconstruction of the *Grown Space Obstacles* in 3D. Fig. 2.6.5 describes the *Grown Space Obstacles* of Workspace D.

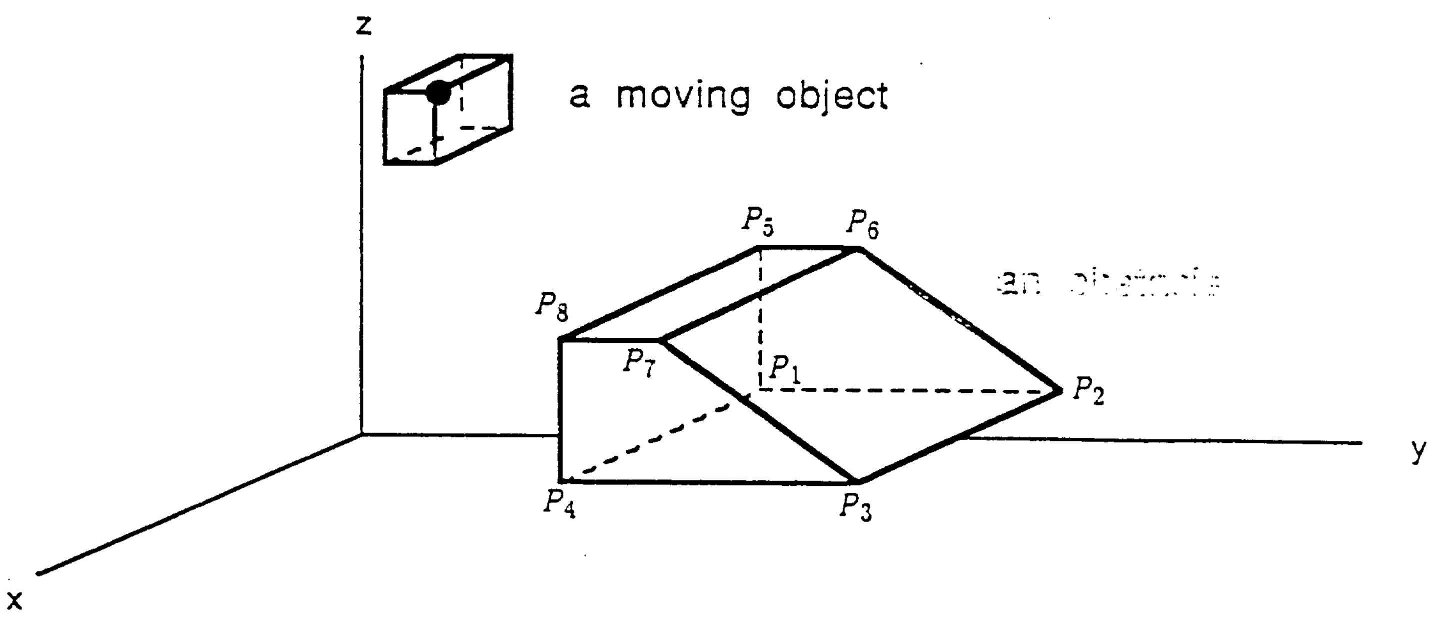

Fig. 2.6.1 A description of Workspace D

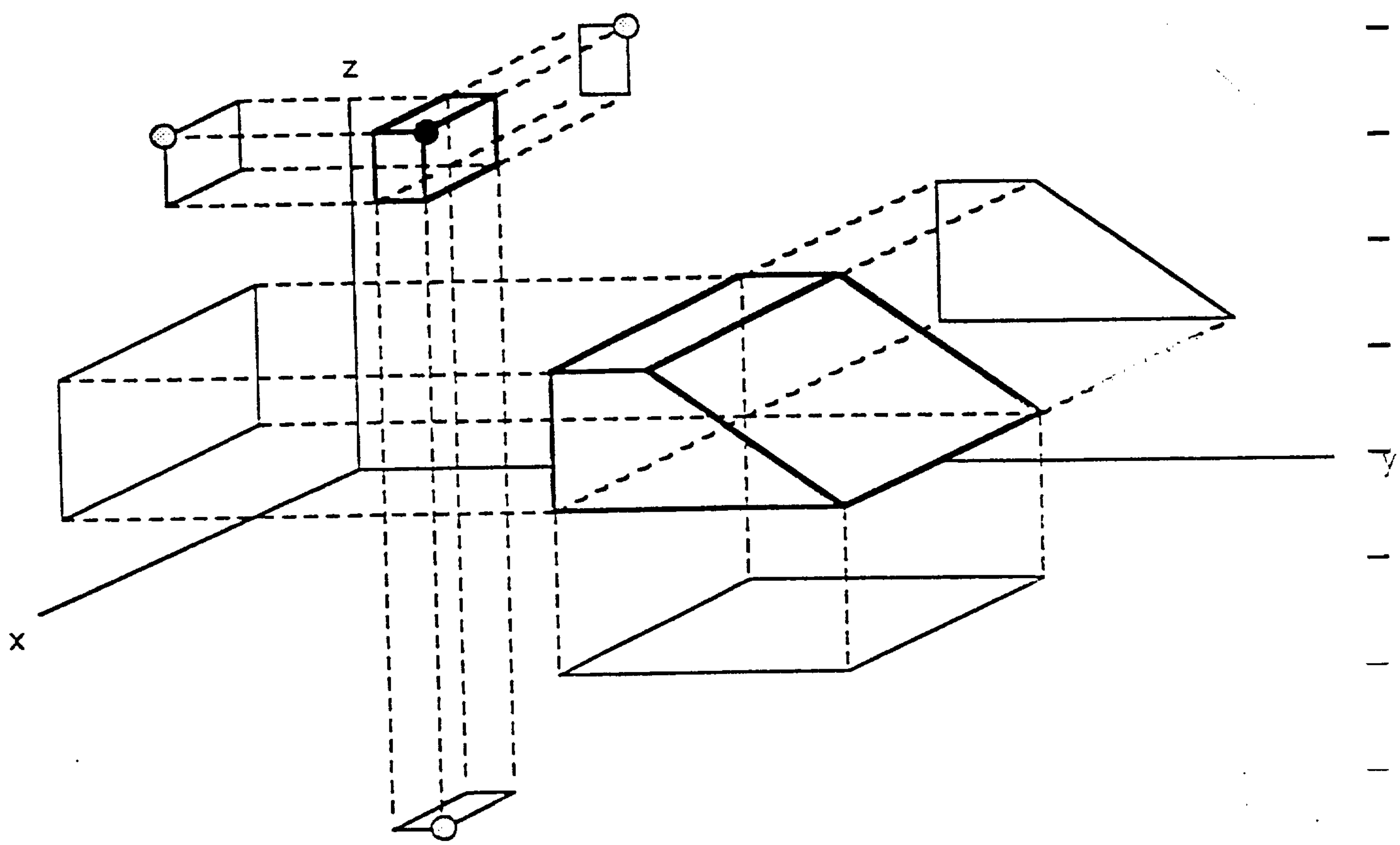

Fig. 2.6.2 A description of three Orthogonal Projections

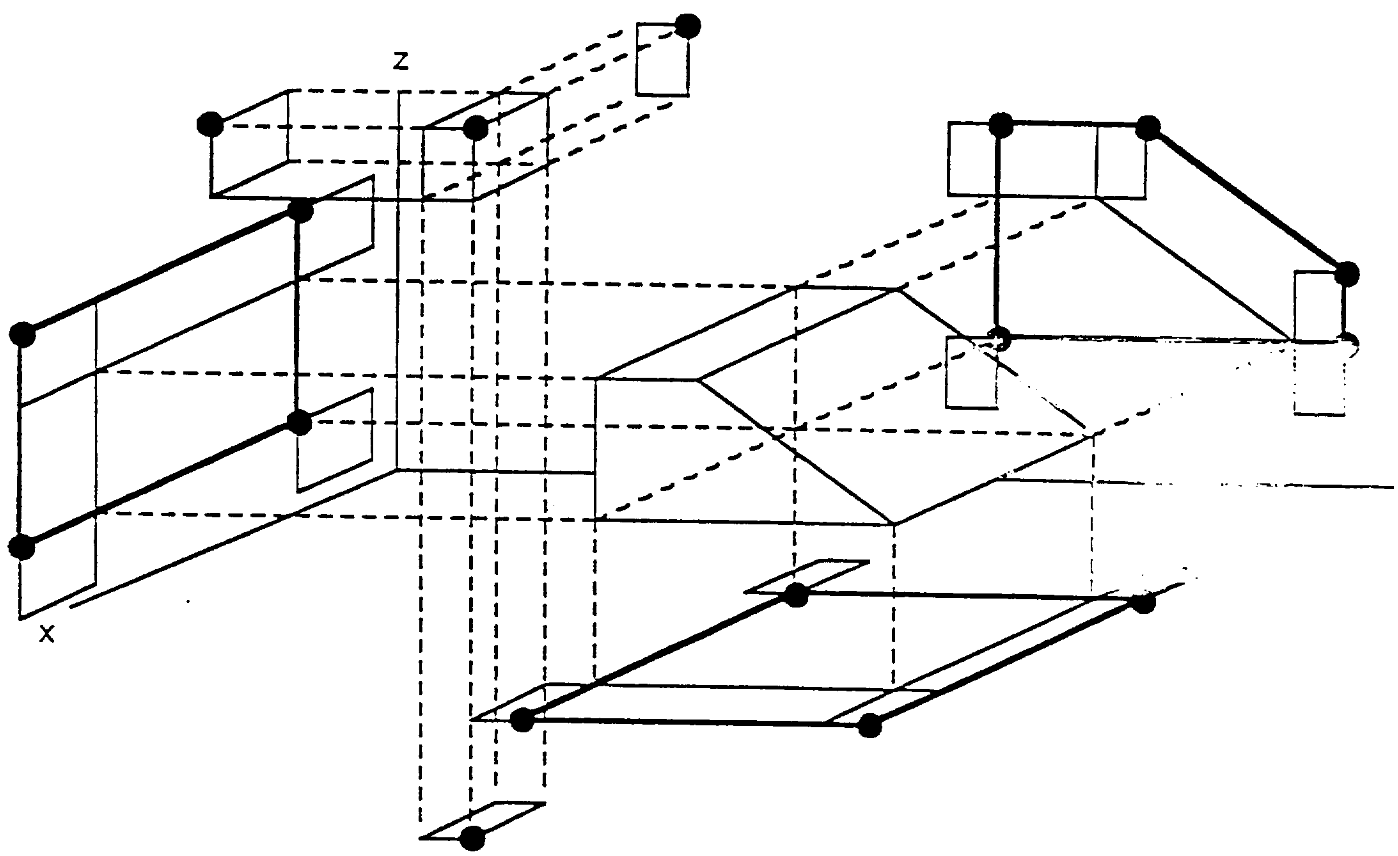

Fig. 2.6.3 A description of Grown Space Obstacles in 2D

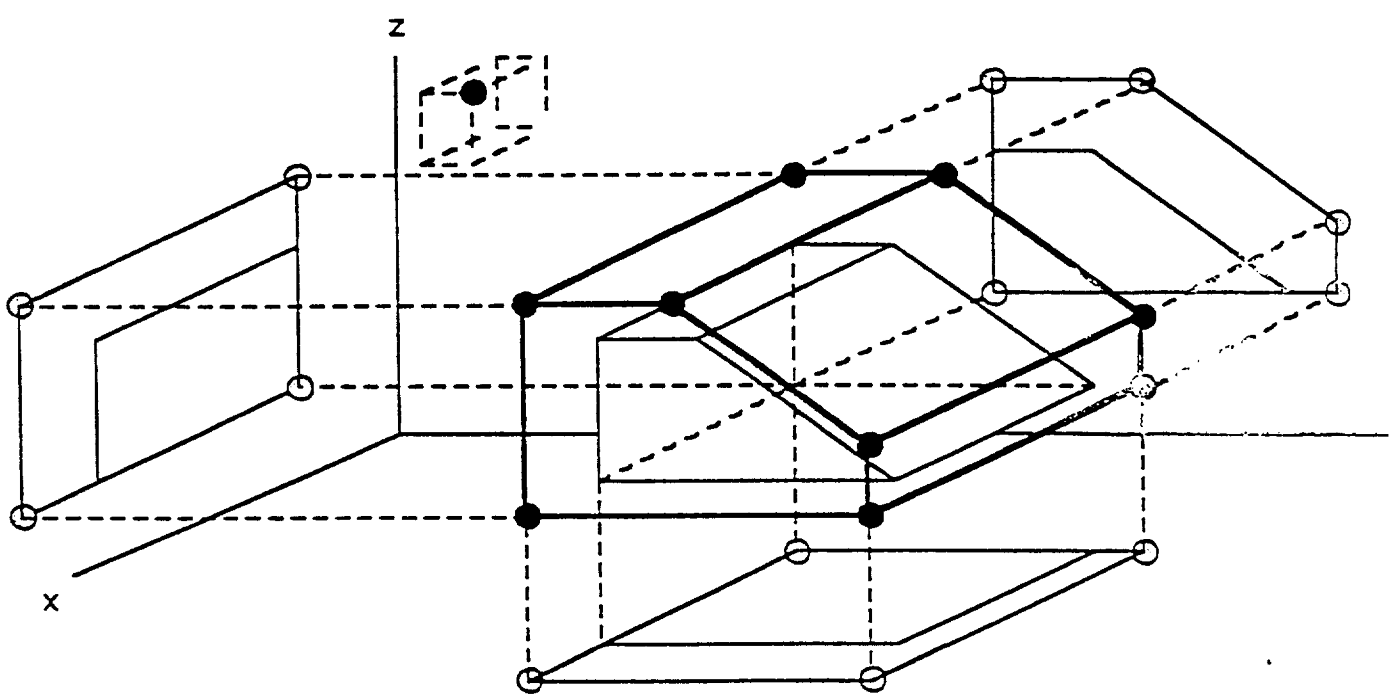

Fig. 2.6.4 A reconstruction of Grown Space Obstacles in 3D

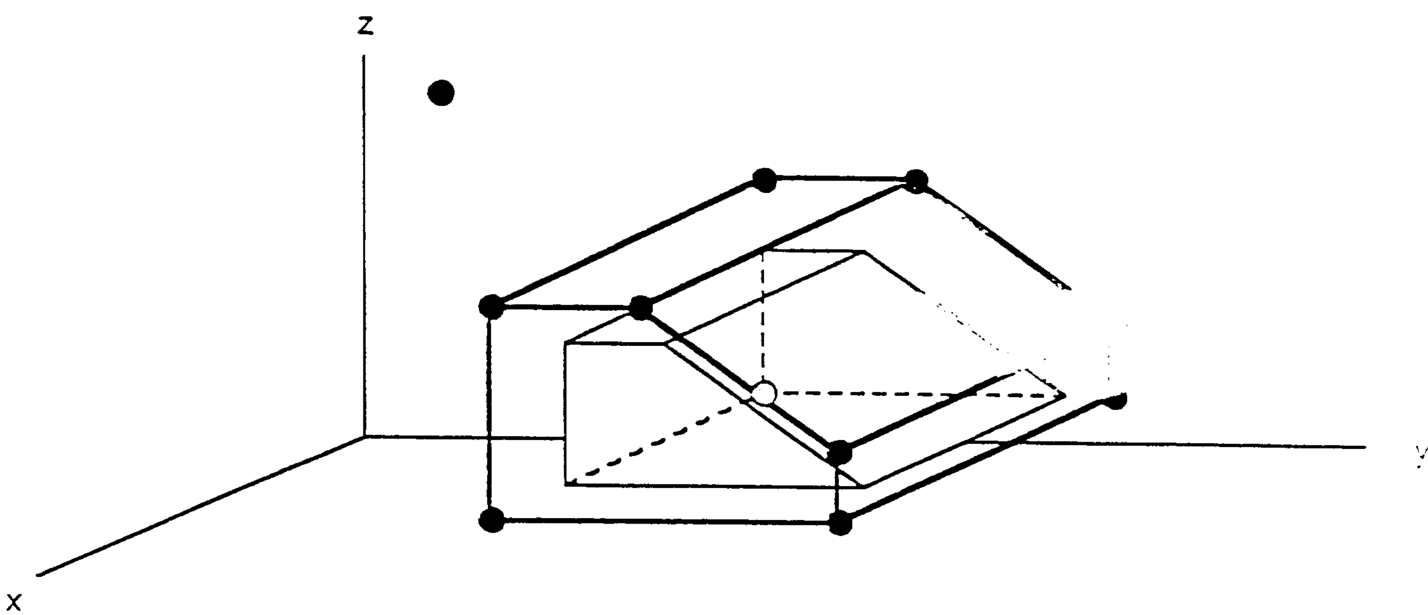

Fig. 2.6.5 The Grown Space Obstacles of Workspace D

2.7 The Recursive Compensation Algorithm

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

Fig. 2.7.1 describes the path calculated by the *VGraph Algorithm* in 3D. The shortest path in 3D may involve going through points on the edges of the *Grown Space Obstacles*. Therefore, the path by *VGraph Algorithm* is not guaranteed to be the shortest path, since the path by the *VGraph Algorithm* involves going through points only on the vertices of the *Grown Space Obstacles*. Fig. 2.7.2 describes the first compensation for the intermediate nodes. Since we assume that all the obstacles are polyhedrals, we are interested in the $z_i(k)$, $i = 1, 2$ and $k = 0, 1, 2 \cdots$. The subscript i indicates the intermediate node and k indicates the number of recursive compensation. This problem is related with minimizing the *Euclidean* distance between two nodes in 3D within some constraints. Fig. 2.7.2 describes the second compensation for the intermediate nodes and Fig. 2.7.3 describes the third compensation for them. Fig. 2.7.4 describes the final path in the *Recursive Compensation Algorithm*, i.e., the better approximation to the shortest path in 3D.

RCA (Recursive Compensation Algorithm)

Procedure RCA(NodeSet, ϵ)
Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \cdots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, $\cdots$, $G(x_n, y_n, z_n)$ The NodeSet is represented by the linked list.
- ϵ = a permissible error
- Function *Distance* will calculate the *Euclidean* distance thro... NodeSet.
- Procedure *Compensate* will find the compensated nodes and will return the set of these nodes.

Begin

    D_1 = Distance(NodeSet)
    Compensate(NodeSet)
    D_2 = Distance(NodeSet)
    If $|D_1 - D_2| < \epsilon$
        Then Return(NodeSet)
        Else RCA(NodeSet, ϵ)

End

Function Distance(NodeSet)
Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \cdots, N_{n-1}\} \cup \{G\}$ search... where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, $\cdots$, ... The NodeSet is represented by the linked list.
- # of NodeSet for *Distance* ≥ 2
- Function *ED* will find the *Euclidean* distance between the first node and the second node in NodeSet.

Begin

    If # of NodeSet = 2
        Then Distance = ED(NodeSet)
        Else Distance = ED(NodeSet)
                + Distance(NodeSet - {*FirstNode*})

End

Procedure Compensate(NodeSet)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \cdots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, $\cdots$, $G(x_n, y_n, z_n)$ The NodeSet is represented by the linked list.
- # of NodeSet for Compensate ≥ 3
- Procedure *Reset* will take the first 3 nodes in NodeSet and replace the second node of the 3 nodes in NodeSet in order to get the set of compensated nodes.

Begin

    If # of NodeSet = 3

        Then Reset(NodeSet)

        Else Reset(NodeSet) $\cup$ Compensate(NodeSet - $\{FirstNode\}$)

End

Procedure Reset(NodeSet)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \cdots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, $\cdots$, $G(x_n, y_n, z_n)$ The NodeSet is represented by the linked list.
- # of NodeSet for *Reset* ≥ 3
- D(d) is the *Euclidean* distance function via 3 nodes in 3D.
- Refer to Appendix C [11] to calculate d to satisfy $\dfrac{\partial D(d)}{\partial d} = 0$.

Begin

    Take out the first 3 nodes in NodeSet.

    Calculate d to satisfy $\dfrac{\partial D(d)}{\partial d} = 0$.

    If d is on the visible edge,

        Then replace the second in NodeSet with the compensated.

End.

[Problem Statement 2.7] Suppose the following vertices are calculated by the *VGraph Algorithm*; $S(3,2,4)$, $N_1(7,4,10)$, $N_2(8,8,9)$, $G(4,11,2)$ shown in Fig. 2.7.1. Calculate the shortest path from the S node to the G node, assuming that the obstacles are polyhedrals.
Define the *convergence ratio* (λ_i)

$$\lambda_i = \frac{y_i(k) - y_i(k+1)}{y_i(k-1) - y_i(k)},$$

where i for intermediate node, k for recursion. Then the *RCA* have the fast convergence ratio from the Theorem 2.7.3. When ϵ is set to 10^{-5}, the *Branch and Bound Method* needs 2301666 miliseconds, however, the *RCA* needs only 416 miliseconds on VAX-11/750. The *Euclidean* distance obtained by the *RCA* is 48% less than that obtained by the *VGraph Algorithm*. The *RCA* is 55,000 times faster than the *Branch and Bound Method* within the same ϵ from the Table 2.7.1. Fig. 2.7.5 describes how fast the *RCA* works. It is proved that the sequences generated by the *RCA* are *Cauchy sequences* by the Theorem 2.7.4. Therefore, the number of recursive compensation could be calculated if ϵ is known. Or ϵ could be calculated if the number of recursive compensation is given. Let's compare the Lozano-Pérez's alleviation method and the *Recursive Compensation Algorithm* to get the same accuracy, ϵ, for the Problem Statement 2.7. Since ϵ is set to 10^{-5}, Lozano-Pérez's alleviation method needs a lot of memory space to store $(2+2\times 8\times 10^5)$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2+2\times 8)$ vertices for the *VGraph*. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time.

Table. 2.7.1 Euclidean distance and Computing time

Algorithm	Distance	Computing time
VGraph	20.3283	0
Branch and Bound	13.7416	2.301.666
RCA	13.7416	416

The simulation of the *RCA* has been done in 3D. See the *Problem Statement 2.7* and the simulation result in 3D. The convergence *RCA* is proved in 2D for simplicity. We are interested in the convergence of the compensated nodes by the *RCA*, i.e., $\{y_1(k)\}$, $\{y_2(k)\}$.

$$
\begin{aligned}
y_1(k) &= y_0 + \frac{y_2(k-1) - y_0}{x_2 - x_0}(x_1 - x_0) \\
&= y_0 + \Delta_0\{y_2(k-1) - y_0\} \\
&= \Delta_0 y_2(k-1) + (1 - \Delta_0)y_0
\end{aligned}
$$

$$\Delta_0 \equiv \frac{x_1 - x_0}{x_2 - x_0}, 0 < \Delta_0 < 1$$

$$
\begin{aligned}
y_2(k) &= y_1(k) + \frac{y_3 - y_1(k)}{x_3 - x_1}(x_2 - x_1) \\
&= y_1(k) + \Delta_1\{y_3 - y_1(k)\} \\
&= (1 - \Delta_1)y_1(k) + \Delta_1 y_3.
\end{aligned}
$$

$$\Delta_1 \equiv \frac{x_2 - x_1}{x_3 - x_1}, 0 < \Delta_1 < 1$$

Definition 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers; we define the *limit superior of* $\{x_n\}$ to be the extended real number

$$\limsup x_n = \inf_{n \geq 1} \sup_{j \geq n} x_j$$

and the *limit inferior of* $\{x_n\}$ to be the extended real number

$$\liminf x_n = \sup_{n \geq 1} \inf_{j \geq n} x_j.$$

Proposition 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers and set

$$s_n^- = \inf_{j \geq n} x_j$$

and

$$s_n^+ = \sup_{j \geq n} x_j.$$

Then for each n the following hold:

i. $s_n^- \leq s_n^+$;

ii. $s_{n+1}^+ \leq s_n^+$;

iii. $s_{n+1}^- \geq s_n^-$;

iv. $\liminf x_n \leq \limsup x_n$.

Proof. Conclusions (i)-(iii) are obvious from the definitions, and so we prove only (iv). Define

$$s^- = \sup s_n^- (= \liminf x_n)$$

and

$$s^+ = \inf s_n^+ (= \limsup x_n).$$

Fix an integer n; observe that if $1 \leq k \leq n$, then by (iii)

$$s_k^- \leq s_n^- \leq s_n^+.$$

On the other hand, if $k > n$, then we may apply (ii) to obtain

$$s_k^- \leq s_k^+ \leq s_n^+.$$

and thus s_n^+ is an upper bound for $\{s_k^-\}$. This implies that $s_n^+ \geq s^-$; but n was arbitrary, and thus s^- is a lower bound for $\{s_n^+\}$, showing that $s^- \leq s^+$, as desired.

Theorem 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers and suppose that $\lim x_n = x_\infty$. If $|x_\infty| < \infty$, then for each $\epsilon > 0$ there is an integer N such that $|x_j - x_\infty| < \epsilon$ whenever $j \geq N$.

Proof. Define $\{s_n^-\}$ and $\{s_n^+\}$ as in Proposition 2.7.1 and fix $\epsilon > 0$. Since $\sup\{s_n^-\} = x_\infty = \inf\{s_n^+\}$, we may choose N_1 and N_2 so large that

$$x_\infty - \epsilon \leq s_{N_1}^-$$

and

$$x_\infty + \epsilon \leq s_{N_2}^+.$$

Setting

$$N = \max\{N_1, N_2\},$$

it follows from Proposition 2.7.1 (ii) and (iii) that for $j \geq N$,

$$x_\infty - \epsilon \leq s_N^- \leq x_j \leq s_N^+ \leq x_\infty + \epsilon,$$

from which $|x_j - x_\infty| \leq \epsilon$ if $j \geq N$.

Definition 2.7.2 [6] [54] Let $\{x_n\}$ be a sequence of real nu... suppose, for each $\epsilon > 0$, there is an index N such that $|x_j$ ... whenever $j, k \geq N$. Then the sequence $\{x_n\}$ is said to be a *Cauchy sequence.*

Theorem 2.7.2 [6] [54] For a sequence $\{x_n\}$ of real numbers, the following are equivalent;

i. $\{x_n\}$ is a *Cauchy sequence*;

ii. $\{x_n\}$ converges to some real number x_∞.

Proof. We first show that (i) $\longrightarrow$ (ii). By Proposition 2.7.1 (iv), it suffices to argue that $\limsup x_n \leq \liminf x_n$. Fix $\epsilon > 0$ and choose N so large that if $j, k \geq N$, then $|x_j - x_k| \leq \epsilon$. Then in particular, $j, k \geq N$, it follows that $x_j \leq x_k + \epsilon$ and hence

$$\inf_{n\geq 1}\sup_{j\geq n} \leq \sup_{j\geq N} x_j \leq x_k + \epsilon.$$

Now recall that $k \geq N$ was arbitrary, and hence

$$\limsup x_n \leq \sup_{n\geq N}\inf_{k\geq n} x_k + \epsilon = \liminf x_n + \epsilon.$$

Since ϵ was arbitrary, this shows that (i) $\longrightarrow$ (ii). For the converse, fix $\epsilon > 0$ and applying Theorem 2.7.1, select an integer N so that $|x_j - x_\infty| \leq \epsilon/2$ if $j \geq N$. Then $j, k \geq N$,

$$|x_j - x_k| \leq |x_j - x_\infty| + |x_\infty - x_k| \leq \epsilon.$$

Theorem 2.7.3 The sequences generated by RCA are mono decreasing.

Proof. The $n^{\underline{th}}$ compensated node $y_1(k)$ can be described by

$$y_1(k) = \Delta_0(1-\Delta_1)y_1(k-1) + (1-\Delta_0)y_0 + \Delta_0\Delta_1 y_3$$

Take $(n+1)^{\underline{th}}$ compensated node $y_1(k+1)$,

$$y_1(k+1) = \Delta_0(1-\Delta_1)y_1(k) + (1-\Delta_0)y_0 + \Delta_0\Delta_1 y_3$$

Subtract one from the other, then

$$y_1(k) - y_1(k+1) = \Delta_0(1-\Delta_1)\{y_1(k-1) - y_1(k)\}$$

$$\frac{y_1(k) - y_1(k+1)}{y_1(k-1) - y_1(k)} = \Delta_0(1-\Delta_1)$$

Since $|\Delta_0(1-\Delta_1)| < 1$,

$$\left|\frac{y_1(k) - y_1(k+1)}{y_1(k-1) - y_1(k)}\right| < 1$$

Therefore, $\{y_1(k)\}$ is mono decreasing.
The $n^{\underline{th}}$ compensated node $y_2(k)$ can be described by

$$y_2(k) = \Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3$$

Take $(n+1)^{\underline{th}}$ compensated node $y_2(k+1)$,

$$y_2(k+1) = \Delta_0(1-\Delta_1)y_2(k) + (1-\Delta_0)(1-\Delta_1)y_0 - \Delta_1 y_3$$

Subtract one from the other, then

$$y_2(k) - y_2(k+1) = \Delta_0(1-\Delta_1)\{y_2(k-1) - y_2(k)\}$$

$$\frac{y_2(k) - y_2(k-1)}{y_2(k-1) - y_2(k)} = \Delta_0(1 - \Delta_1)$$

Since $|\Delta_0(1 - \Delta_1)| < 1$,

$$\left|\frac{y_2(k) - y_2(k+1)}{y_2(k-1) - y_2(k)}\right| < 1$$

Therefore, $\{y_2(k)\}$ is mono decreasing.

QED.

Theorem 2.7.4 The sequences generated by *RCA* are *Cauchy sequences.*

Proof. This theorem can be proved by the *mathematical induction.* By the Theorem 2.7.3, we can get

$$\frac{y_1(2)}{y_1(1)} < 1$$

Substitute $y_1(2) = \Delta_0(1 - \Delta_1)y_1(1) + \Delta_0\Delta_1 y_3 + (1 - \Delta_0)y_0$

$$\frac{\Delta_0(1 - \Delta_1)y_1(1) + \Delta_0\Delta_1 y_3 + (1 - \Delta_0)y_0}{y_1(1)} < 1$$

$$\{1 - \Delta_0(1 - \Delta_1)\}y_1(1) > (1 - \Delta_0)y_0 + \Delta_0\Delta_1 y_3$$

(i) Since $1 - \Delta_0(1 - \Delta_1) > 0$, we can get the following inequality.

$$y_1(1) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

(ii) Assume that

$$y_1(k) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

(iii) We should prove that

$$y_1(k+1) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

If the above inquality is hold for any k, the sequence generated by *RCA* is a *Cauchy sequence* because the sequence by *RCA* is mono decreasing by the Theorem 2.7.3.

$$y_1(k-1) = \Delta_0(1 - \Delta_1)y_1(k) - (1 - \Delta_0)y_0 + \Delta_0\Delta_1 y_3$$

Substitute $y_1(k)$ with the inequality from (ii), then

$$y_1(k+1) > \Delta_0(1-\Delta_1)\{\frac{1-\Delta_0}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3\}$$
$$+(1-\Delta_0)y_0 + \Delta_0\Delta_1 y_3$$

Therefore,

$$y_1(k+1) > \frac{1-\Delta_0}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

By the Theorem 2.7.3, we can get

$$\frac{y_2(2)}{y_2(1)} < 1$$

Substitute $y_2(k) = \Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3$

$$\frac{\Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3}{y_1(1)} < 1$$

$$\{1-\Delta_0(1-\Delta_1)\}y_2(1) > (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3$$

(iv) Since $1-\Delta_0(1-\Delta_1) > 0$, we can get the following inequality.

$$y_2(1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

(v) Assume that

$$y_2(k) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

(vi) We should prove that

$$y_2(k+1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

If the above inquality is hold for any k, the sequence generated by *RCA* is a *Cauchy sequence* because the sequence by *RCA* is mono decreasing by the Theorem 2.7.3.

$$y_2(k) = \Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3$$

Substitute $y_2(k)$ with the inequality from (v), then

$$y_2(k+1) > \Delta_0(1-\Delta_1)\{\frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3\}$$
$$+(1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1 y_3$$

Therefore,

$$y_2(k+1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

Therefore, the sequences generated by *RCA* are *Cauchy sequences* by the Theorem 2.7.2 and Theorem 2.7.3, because $\{y_1\}$ and $\{y_2\}$ converge to some real numbers and they are mono decreasing.

QED.

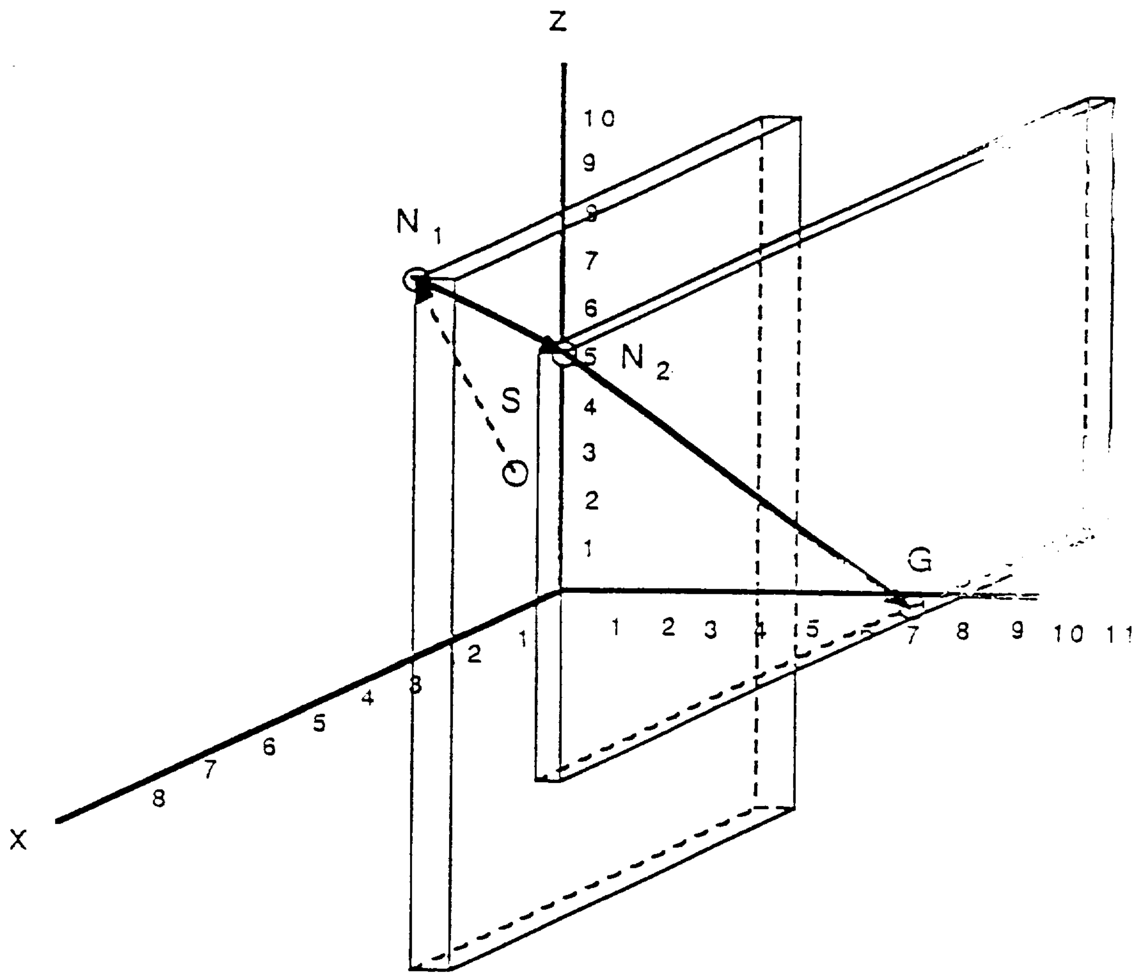

Fig. 2.7.1 The path calculated by the VGraph Algorithm

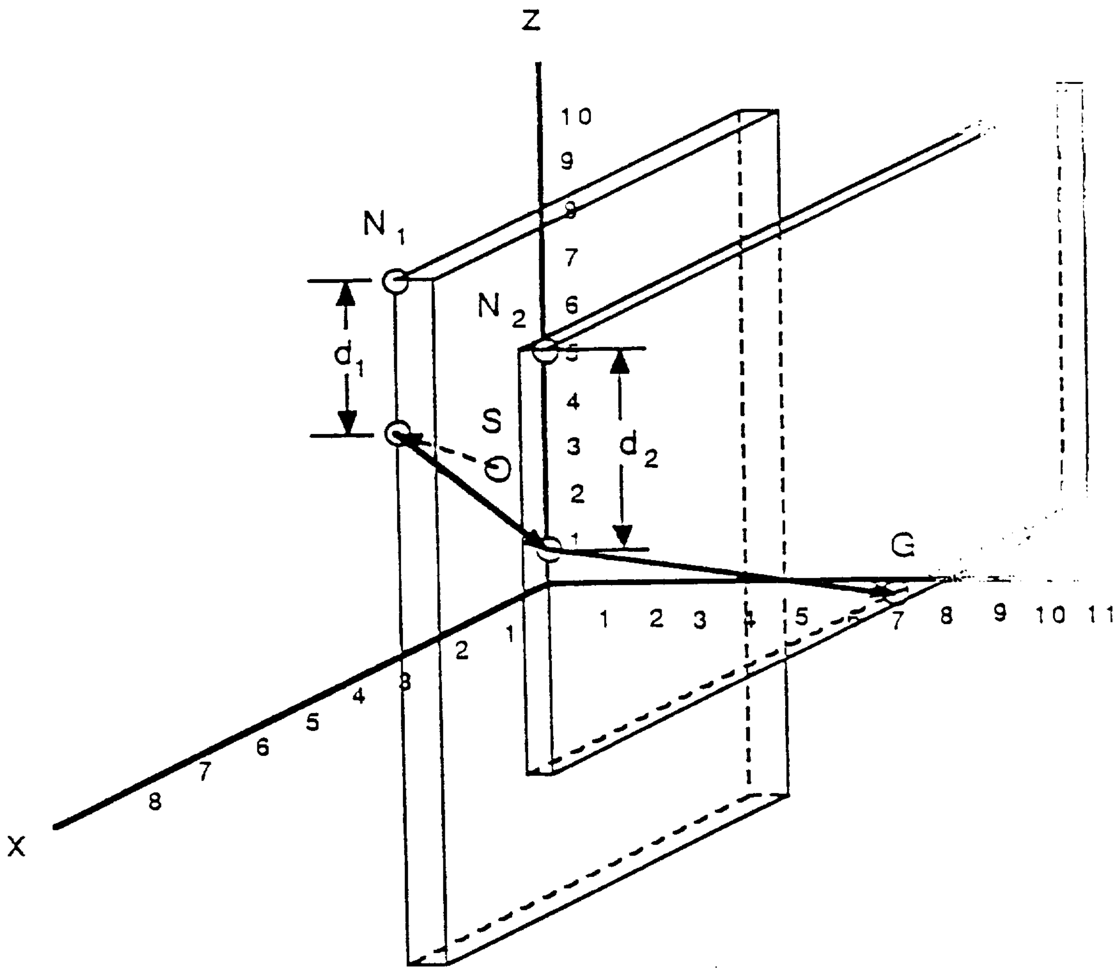

Fig. 2.7.2 The first compensation by the RCA

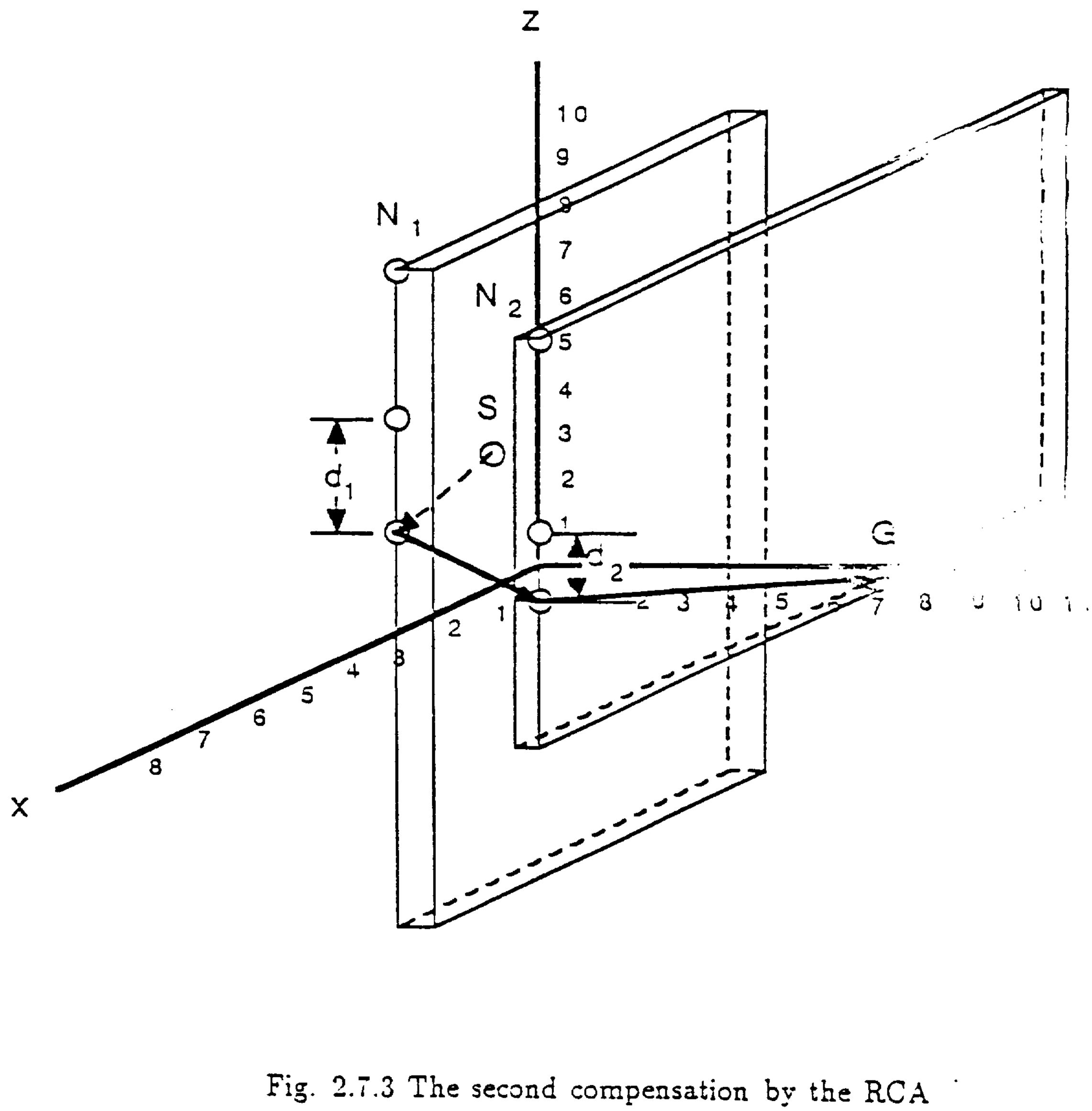

Fig. 2.7.3 The second compensation by the RCA

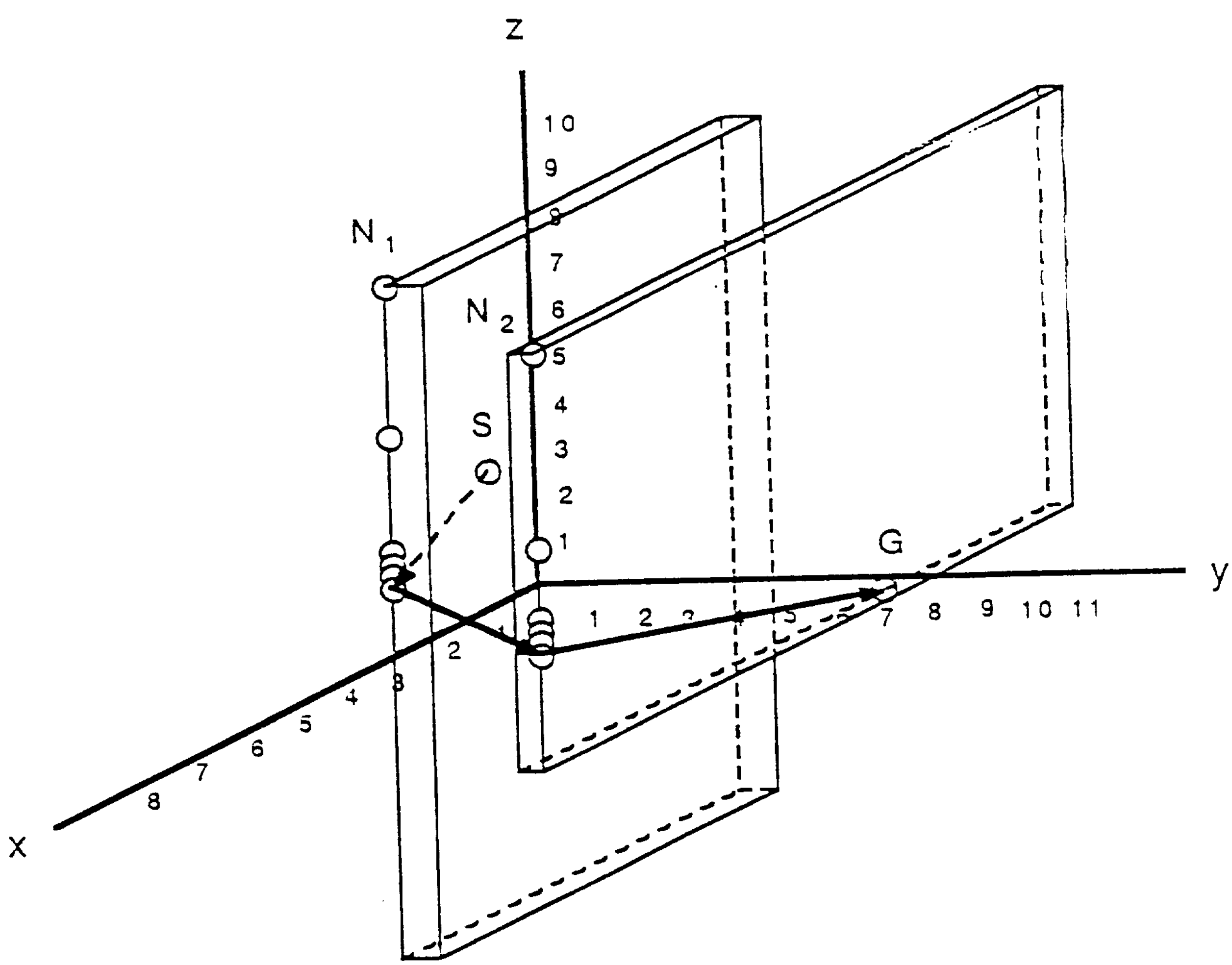

Fig. 2.7.4 The final path by the RCA

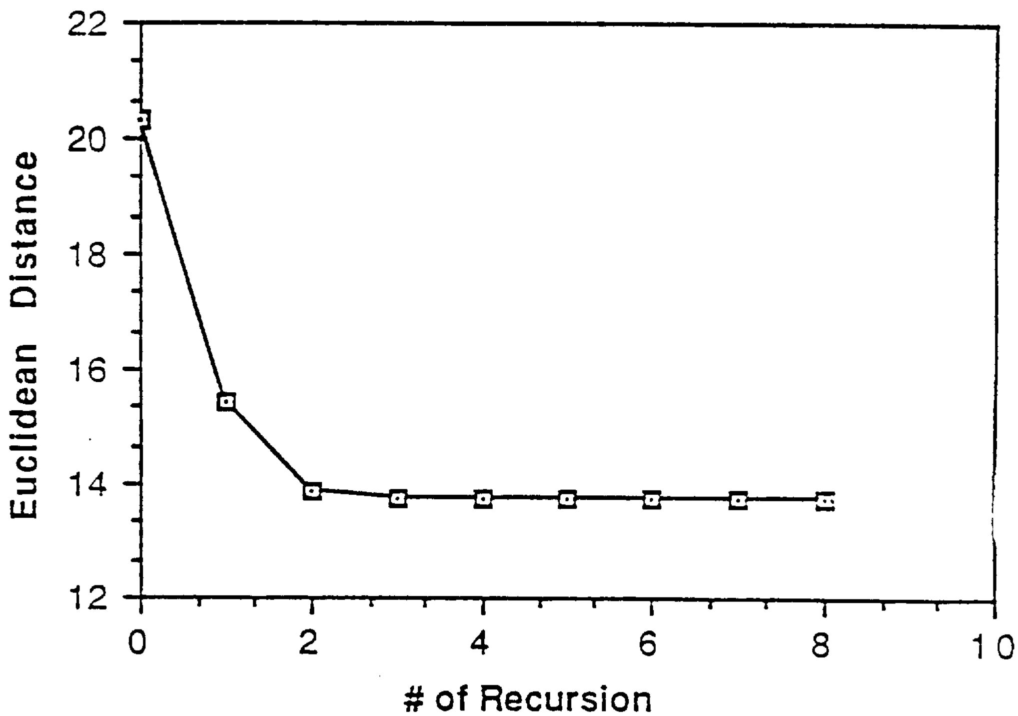

Fig. 2.7.5 The Euclidean distance by the RCA

3. Problem Statement, Preliminary Results and Proposed Work

3.1 Problem Statement

Consider the *VGraph Algorithm* for a moving object to find the collision-free shortest path in a workspace with some obstacles. A lot of work has been done in this field, which has the following design steps:

- Build the *Grown Space Obstacles.*
- Find the visible vertices by detecting interferences.
- Build the *VGraph* with a set of the visible vertices.
- Search the *VGraph* by the graph search algorithm.

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez [29] points out the drawbacks of the *VGraph Algorithm.* The first drawback is related with the rotation of a moving object. Since the *VGraph Algorithm* require moving a object along obstacle boundaries, shortest paths are very susceptible to inaccuracies in the object models. This drawback can be solved by using the *sliced projection method* [28] [29] [30]. However, the *VGraph Algorithm* has serious drawbacks [29] when the obstacles are three-dimensional:

- shortest paths do not typically traverse the vertices of the *Grown Space Obstacles,*
- there may be no paths via vertices, within the enclosing polyhedral region R, although other types of safe paths within R may exist.

Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles.* However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph,* which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph.* The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph.* However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. So, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D. Therefore, a new algorithm, called the *Extended VGraph Algorithm,* should deal with the drawbacks of the *VGraph Algorithm.*

The Extended VGraph Algorithm has the following design steps:

1) Apply the *Orthogonal Projection Method* to get the *Grown Space Obstacles* in 3D.
   i) Project obstacles in 3D onto the projection spaces.
   ii) Build the Grown Space Obstacles in 2D.
   iii) Select the necessary *Grown Space Obstacles* for the *VGraph.*
   iv) Reconstruct the Grown Space Obstacles in 3D.
2) Find the visible vertices by deteccing interferences.
3) Build the *VGraph* with a set of the visible vertices.
4) Search the *VGraph* by the graph search algorithm.
5) Apply the *Recursive Compensation Algorithm* to obtain the collision-free shortest path in 3D.

3.2 Preliminary Results

- The *VGraph Algorithm* has been implemented to find the collision-free shortest path in two dimensional space. This *VGraph Algorithm* can deal with not only translations of a moving object but also its rotations by using the θ sliced projection. For the simulation of the *VGraph Algrothm,* see the Problem Statement 3.2.1 and the Problem Statement 3.2.2.
- The *Orthogonal Projection Method* has been implemented to build the *Grown Space Obstacles* in 3D and to represent them in three projected two-dimensional spaces. Since the *Orthogonal Projection Method* avoids building the unnecessary *Grown Space Obstacles,* it can make the *VGraph* simpler than any other algorithms that use all of the *Grown Space Obstacles.* Therefore, the *Orthogonal Projection Method* can save the memory space to store the representation of the *Grown Space Obstacles* and it can shorten the graph search time because of the simpler *VGraph.* For the simulation

of the *Orthogonal Projection Method*, see the Problem Statement 3.2.3.

- The *Recursive Compensation Algorithm* has been implemented to find the collision-free shortest path in 3D. The *Recursive Compensation Algorithm* can guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*. The property of convergency of the *Recursive Compensation Algorithm* is proved by the Theorem 2.7.4. Since ϵ is set to 10^{-5}, Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 8 \times n \times \epsilon^{-1})$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 8 \times n)$ vertices for the *VGraph*. The accuracy is defined by ϵ whose value is very small and n is the number of obstacles in workspace. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time. For the simulation of the *Recursive Compensation Algorithm*, see the Problem Statement 3.2.4.
- The *Extended VGraph Algorithm* has been presented to solve the drawbacks of the *VGraph Algorithm*. Each module of the *Extended VGraph Algorithm* has been implemented in the Problem Statement 3.2.2, the Problem Statement 3.2.3 and the Problem Statement 3.2.4.

[Problem Statement 3.2.1] Consider the problem, shown in Fig. 2.2.1, assuming that the objects are polyhedrons and their visual informations are available and they are represented by vertices. Find the collision-free shortest distance from *Start* to *Goal* with $\frac{\pi}{4}$ sliced rotation.

Table 3.2.1 Simulation result of the VGraph algorithm.

The shortest path is calculated by the VGraph Algorithm.			
Start Node = 1, Goal Node = 27,			
Path represented by internal nodes: $1 \longrightarrow 16 \longrightarrow 6 \longrightarrow 27$			
From	To	Cost	Rotation
Start in 0 sliced	A_3 in $\frac{\pi}{2}$ sliced	4.125	$\frac{\pi}{2}$
A_3 in $\frac{\pi}{2}$ sliced	B_1 in $\frac{\pi}{2}$ sliced	5.025	0
B_1 in $\frac{\pi}{2}$ sliced	Goal in 0 sliced	4.031	$-\frac{\pi}{2}$
The total cost between Start and Goal = 13.179			

Fig. 2.4.2 shows the collision-free shortest path with $\frac{\pi}{4}$ sliced rotation by the *VGraph Algorithm*. The path with $\frac{\pi}{4}$ sliced rotation has 13.179 *Euclidean* distance, while the path without sliced rotation has 25.452 *Euclidean* distance. The path segment with sliced rotation is described in the Table 3.2.1, the path segment without sliced rotation

is $\{Start - C_1 - B_3 - B_2 - Goal\}$. Hence, the sliced rotation of the moving object can shorten the *Euclidean* distance. However, there is a trade off between accuracy and speed. If the small sliced rotation is considered, then the better approximation to the shortest path can be obtained, but more memory space to store each *VGraph* is needed. The result for the Problem Statement 3.2.1 comes from the file [PATH] in Appendix B. The programming list of this simulation is available in the Appendix A.

[Problem Statement 3.2.2] Assuming that the horizontal length of the moving object is 2, its vertical length is 1 and θ is $\frac{\pi}{6}$ and obstacles are given as in Fig. 2.3.5, draw the rotational *GSpace Obstacles*.

Fig. 2.3.6 - Fig. 2.3.11 draw the rotational *GSpace Obstacles*. The Problem Statement 3.2.2 shows that the *VGraph Algorithm* can handle the rotation of a moving object by the θ sliced projection method. The result of the Problem Statement 3.2.2 comes from the file [ROTATION] in Appendix D. The programming list for this simulation is available in Appendix C.

[Problem Statement 3.2.3] Build the *Grown Space Obstacles* in 3D by using the *Orthogonal Projection Method*, assuming that the object in 3D is a polyhedron, shown in Fig. 2.6.1. The object has the following vertices; $P_1(x_1, y_1, z_1)$, $P_2(x_1, y_2, z_1)$, $P_3(x_2, y_2, z_1)$, $P_4(x_2, y_1, z_1)$, $P_5(x_1, y_1, z_2)$, $P_6(x_1, y_2, z_2)$, $P_7(x_2, y_2, z_2)$, $P_8(x_2, y_1, z_2)$, where $x_1 = 7, y_1 = 5, z_1 = 3, x_2 = 14, y_2 = 10, z_2 = 12$.

Fig. 2.6.2 describes three Orthogonal Projections of Workspace D. Fig. 2.6.3 describes the *Grown Space Obstacles* in 2D. Fig. 2.6.4 describes the reconstruction of the *Grown Space Obstacles* in 3D. Fig. 2.6.5 describes the *Grown Space Obstacles* of Workspace D. The result for the Problem Statement 3.2.3 comes from the file [PROJECTION] in Appendix J. The programming list for this simulation is available in Appendix I.

[Problem Statement 3.2.4] Suppose the following vertices are calculated by the *VGraph Algorithm*; $S(3, 2, 4)$, $N_1(7, 4, 10)$, $N_2(8, 8, 9)$, $G(4, 11, 2)$ shown in Fig. 2.7.1. Calculate the shortest path from the S node to the G node, assuming that the obstacles are polyhedrals.

When ϵ is set to 10^{-5}, the *Branch and Bound Method* needs 2301666 miliseconds, however, the *RCA* needs only 416 miliseconds on VAX 11/750. The *Euclidean* distance obtained by the *RCA* is 48% less than that obtained by the *VGraph Algorithm*. The *RCA* is 55,000 times faster than the *Branch and Bound Method* within the same ϵ from the Table 3.2.4. Fig. 2.7.5 describes how fast the *RCA* works. It is proved that the sequences generated by the *RCA* are *Cauchy sequences* by the

Theorem 2.7.4. Therefore, the number of recursive compensation could be calculated if ϵ is known. Or ϵ could be calculated if the number of recursive compensation is given. Let's compare the Lozano-Pérez's alleviation method and the *Recursive Compensation Algorithm* to get the same accuracy, ϵ, for the Problem Statement 3.2.4. Since ϵ is set to 10^{-5}, Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 2 \times 8 \times 10^5)$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 2 \times 8)$ vertices for the *VGraph*. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time. The result of the Problem Statement 3.2.4 comes from the file $[BB_{output}]$ in Appendix F and the file $[RCA_{output}]$ in Appendix H. The programming list of this simulation is available in the Appendix E and Appendix G.

Table. 3.2.4 Euclidean distance and Computing time

Algorithm	Distance	Computing time
VGraph	20.3283	0
Branch and Bound	13.7416	2.301.666
RCA	13.7416	416

3.3 Proposed Work

- Improve the θ sliced projection method by an algorithm to select the proper θ.
- Simplify the *VGraph* of the polygon by the *Convex Rope Algorithm*.
- Solve the collision-free Findpath problem for the dynamic obstacles.
- Exchange the knowledge on the *path planning* with other coordinators for the intelligent robot control.
- Compare the *Extended VGraph Algorithm* with other algorithms.

REFERENCES

1. Aho, A. V. and Ullman, J. D., "Data Structures and Algorithms," *Addison-Wesley*, Reading, Mass., 1983.
2. Akman, Varol, "Shortest Paths Avoiding Polyhedral Obstacles in 3-Dimensional Euclidean Space," *Ph.D. dissertation*, Dep. Computer and System Eng., Rensselaer Polytechnic Institute, 1985.
3. Aldefeld, B., "On Automatic Recognition of 3D Structures from 2D Representation," *Computer Aided Design*, Vol. 15, No. 2, March 1980, pp. 59-64.
4. Ahuja, N. and Chien, R. T. and Yen, R. and Bridwel, N., "Interference Detection and Collision Avoidance among Three Dimensional Objects," *1st Ann. Nat. Conf. Artificial Intelligence*, Stanford Univ., Stanford, CA, August 1980, pp. 44-48.
5. Bajaj, Chanderjit, "An Efficient Parellel Solution for Euclidean Shortest Path in Three Dimensions," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1897-1900.
6. Barte, Robert G. and Sherbert, Donald R., "Introduction to Real Analysis," *John Wiley* 1982.
7. Boyse, J. W., "Interference Detection among Solids and Surfaces," *Communications of the ACM*, Vol. 22, No. 1, January 1979, pp. 3-9.
8. Brooks, Rodney A., "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-13, No. 3, March/April 1983, pp. 190-197.
9. Brooks, Rodney A., "Planning Collision-Free Motions for Pick-and-Place Operations," *The International Journal of Robotics Research*, Vol. 2, No. 4, Winter 1983, pp. 19-44.
10. Brooks, Rodney A. and Lozano-Perez, Tomas, "A Subdivision Algorithm in Configuration Space for Find-Path with Rotation," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-15, No. 2, March/April 1985, pp. 224-233.
11. Chung, C. H. and Saridis, G. N., "An Obstacle Avoidance Motion Organizer for an Intelligent Robot," *Technical Report RAL-TR-88-117*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
12. Clocksin, W. F. and Mellish, C. S., "Programming in Prolog," *Springer-Verlag*, New York, 1987.
13. Davis, R. H. and Camacho, M., "The Application of Logic Programming to the Generation of Paths for Robots," *Robotica*, Vol.

2, 1984, pp. 93-103.

14. Dupont, Pierre E. and Derby, Stephen, "Planning Collision Free Paths for Redundant Robots Using a Selective Search of Configuration Space," *ASME Mechanisms Conference 86*, 1986.

15. Dupont, Pierre E., "Planning Collision Free Paths for Kinematically Redundant Robots by Selectively Mapping Configuration Space," *Ph.D dissertation,* Dep. Mech. Eng., Rensselaer Polytechnical Institute, 1988.

16. Faverjon, Bernard, "Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator," *in Proc. IEEE Int. Conf. Robotics*, Atlanta, GA, March 1984, pp. 504-512.

17. Fu, K. S. and Conzalez, R. C. and Lee, C. S. G., "Robotics: Control, Sensing, Vision and Intelligence," *McGraw-Hill Book Company*, 1987.

18. Gilbert, Elmer G. and Johnson, Daniel W., "The Application of Distance Functions to the Optimization of Robot Motion in the Presence of Obstacles," *in Proc. 23rd Conf. Decision and Control*, December 1984, pp. 1338-1344.

19. Gilbert, Elmer G., "Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles," *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 1, March 1985, pp. 21-30.

20. Gilbert, Elmer G. and Johnson, D. W. and Keerthi, S. S., "A Fast Procedure for Computing the Distance between Complex Objects in Three Dimensional Space," *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, April 1988, pp. 193-203.

21. Gouzènes, Laurent, "Strategies for Solving Collision Free Trajectories Problems for Mobile and Manipulator Robots," *The International Journal of Robotics Research*, Vol. 3, No. 4, Winter 1984, pp. 51-65.

22. Hart, P. and Nilsson, N. J. and Raphael, B. A., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybernetics*, Vol. SSC-4, No. 2, July 1980, pp. 100-107.

23. Hayward, Vincent, "Fast Collision Detection Scheme by Recursive Decomposition of a Manipulator Wrokspace," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1044-1049.

24. Herman, Martin, "Fast, Three-Dimensional, Collision Free Motion Planning," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1056-1063.

25. Hunter, Gregory M. and Steiglitz, Kenneth, "Operations on Images Using Quad Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-1, No. 2, April 1979, pp. 145-153.

26. Khatib, Oussama, "Real-Time Obstacle Avoidance for Manipu-

lators and Mobile Robots," *The International Journal of Robotics Research*, Vol. 5, No. 1, Spring 1986, pp. 90-98.

27. Lawler, E. L. and Wood D. E., "Branch and Bound Methods: A Survey," *Operations Research*, July-August 1966, pp. 699-719.

28. Lozano-Pérez Tomás and Wesley, Michael A., "An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles," *Communications of the ACM*, Vol. 22, No. 10, October 1979, pp. 560-570.

29. Lozano-Pérez Tomás, "Automatic Planning of Manipulator Transfer Movements," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-11, No. 10, October 1981, pp. 681-698.

30. Lozano-Pérez Tomás, "Spatial Planning: A Configuration Space Approach," *IEEE Trans. Computers*, Vol. C-32, No. 2, Feburary 1983, pp. 108-120.

31. Lozano-Pérez Tomás and Jones, Joseph L. and Mazer, Emmanuel and O'Donnel, Patrick A. and Grimson, L., "Handy; A Robot System that Recognizes, Plans, and Manipulates," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1987, pp. 843-849.

32. Lozano-Pérez Tomás, "A Simple Motion-Planning Algorithm for General Robot Manipulators," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987, pp. 224-238.

33. Luh, J. Y. S. and Campbell, C. E., "Collision-Free Path Planning for Industrial Robots," *in Proc. 21th IEEE Conf. Decision and Control*, 1982, pp. 84-88.

34. Lumelsky, Vladimir J., "Continuous Motion Planning in Unknown Environment for a 3D Cartesian Robot Arm," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1050-1055.

35. Lumelsky, Vladimir J., "Effect of Kinematics on Motion Planning for Planar Robot Arms Moving amidst Unknown Obstacles," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987, pp. 207-223.

36. Luo, G. L. and Saridis, George N., "Optimal/PID Formulation for Control of Robotic Manipulators," *Tech. Report RAL-TR-84-034*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.

37. Luo, G. L. and Saridis, George N., "L-Q Design of PID Controllers for Robot Arms," *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 3, Sep. 1985, pp. 152-159.

38. Mangasarian, O. L. and Meyer, R. R. and Robinson, S. M., "Nonlinear Programming 2," *The Academic Press*, New York, 1975.

39. Maron, Melvin J., "Numerical Analysis," *Macmillan Publishing Co., Inc.* 1982.

40. Martelli, A. and Motanari, U., "From Dynamic Programming to

Search Algorithms with Functional Costs," *Proc. 4th International Joint Conference on Artificial Intelligence*, 1975, pp. 345-350.

41. Martelli, A., "On the Complexity of Admissible Search Algorithms," *Artificial Intelligence*, 1977, pp. 1-13.
42. Maruyama, K., "A Procedure to determine intersections between polyhedral objects," *Int. J. of Computer and Information Science*, 1972, pp. 255-266.
43. Meagher, Donald, "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer," *Technical Report IPL-TR-80-111*, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
44. Meagher, Donald, "Octree Generation, Analysis and Manipulation," *Technical Report IPL-TR-82-027*, Image Processing laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
45. Meyer, Walter and Benedict, Powell, "Path Planning and the Geometry of Joint Space Obstacles," *in Proc. IEEE Int. Conf. Robotics and Automation*, 1988, pp. 215-219.
46. Nilsson, N., "A Mobile Automation: An Application of Artificial Intelligence Technique," *in Proc. 1st International Joint Conference on Artificial Intelligence*, 1969, pp. 509-520.
47. Nilsson, Nils J., "Principle of Artificial Intelligence," *Tioga Publishing Company*, Palo Alto, CA, 1980.
48. Noborio, Hiroshi and Fukuda, Shozo and Arimoto, Suguru, "Construction of the Octree Approximating Three-Dimensional Objects by Using Multiple Views," *IEEE Trans. on PAMI*, Vol. 10, No. 6, November 1988, pp. 769-782.
49. Paul, R. P., "Manipulator Cartesian Path Control," *IEEE Trans. Syst., Man, Cybern.*, Vol. SMC-9, Nov. 1979, pp. 702-711.
50. Paul, R. P., "Robot Manipulators: Mathematics, Programming, and Control," *The MIT Press*, Cambridge, Massachusetts, 1982.
51. Paul, R. P. and Zhang, H, "The Dynamics of the PUMA Manipulator," *in Proc. 1983 American Control Conf.*, pp. 491-496.
52. Pearl, Judea, "Heuristics: Intelligent Search Stategies for Computer Problem Solving," *Addition-Wesley Publishing Company*, 1984.
53. Peshkin, Michael A. and Sanderson, Arthur C., "Reachable Graps on a Polygon: The Convex Rope Algorithm," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986, pp. 53-58.
54. Ray, William O., "Real Analysis," *Prentice Hall*, 1988.
55. Saridis, George N., "Intelligent Robotic Control," *IEEE Trans. Au-*

tomatic Control, Vol. AC-28, No. 5, May 1983, pp. 547-557.

56. Saridis, George N. and Graham, J. H., "Linguistic Decision Schemata for Intelligent Robots," *Automatica*, Vol. 20, No. 1, 1984, pp. 121-126.
57. Sedgewick, Robert, "Algorithms," *Addison-Wesley Publishing Company*, 1983.
58. Tarn, T. J. and Bejczy, A. K. and Han, Shuotiao and Yun, Xiaoping, "Inertia Parameters of PUMA 560 Robot Arm," *Technical Report SSM-RL-85-01*, Robotics Laboratory, Department of Systems Science and Mathematics, Washington University, St. Louis, Missouri, 63130.
59. Udupa, S. M., "Collision Detection and Avoidance in Computer Controlled Manipulators," *in Proc. 5th International Joint Conference on Artificial Intelligence*, MIT, Cambridge, MA, August 1977, pp. 737-748.
60. Valavanis, Kimon P. and Saridis, George N., "A Mathematical Formulation for the Analytical Design of Intelligent Machines," *Technical Report RAL-TR-86-085*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
61. Valentí, Joan Ilari i, "Study of New Heuristics to Compute Collision Free Paths of Rigid Bodies in a 2D Universe," *Ph.D. dissertation*, Dep. D'enginyeria Cibernètica, Universitat Politècnica De Catalunya, 1987.
62. Winston, P. A., "Artificial Intelligence," *Addison-Wesley Publishing Company*, 1984.
63. Wong, E. K. and Fu, K. S., "A Hierachical Orthogonal Space Approach to Three Dimensional Path Planning," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986, pp. 42-53.

Appendix A: Simulation of the VGraph Algorithm

```
program VGA(OBSTACLES,GSPACE,VERTICES,PATH,output);
{________________________________________________________________

    Author  :  C. H. Chung

    Version :  4.7

    Date    :  December 1, 1988

 ________________________________________________________________

    This program is designed to simulate the VGraph Algorithm.

    It consists mainly of 3 procedures.

       1.  BuildGSpaceObstacles(List).
                .  To build the Grown Space Obstacles
                .  INPUT  FILE : OBSTACLES, (GSPACE)
                .  OUTPUT FILE : output, GSPACE
                .  output of this procedure : List

       2.  BuildVGraph(A, List).
                .  To build the VGraph
                .  The first part of this procedure mainly consists
                   of the Interference Checking, i.e. the Visible
                   Vertices, and the second part of this procedure
                   mainly consists of the VGraph Construction.
                .  INPUT  FILE : _______
                .  OUTPUT FILE : VERTICE
                .  input  of this procedure : List
                .  output of this procedure : A

       3.  SearchVGraph(A, LinkedPath).
                .  To find the shortest path of the VGraph
                .  LinkedPath holds the information of the shortest
                   path by the VGraph Algorithm.
                .  INPUT  FILE : ____
                .  OUTPUT FILE : PATH
                .  input  of this procedure : A
                .  output of this procedure : LinkedPath



     Pay a special attentation on the data structure of List.
     List consists of S1, S2,
                      A1(1), A2(1), B1(1), B2(1), C1(1), C2(1),
                      A1(2), A2(2), B1(2), B2(2), C1(2), C2(2),
                      A1(3), A2(3), B1(3), B2(3), C1(3), C2(3),
                      A1(4), A2(4), B1(4), B2(4), C1(4), C2(4),
                      G1, G2.
```

```
    ________________________________________________________________}

type
   PointType = record
                  x, y : real;
               end;
   VerticeType = ^Nodes;
   Nodes = record
              Node : PointType;
              Next : VerticeType;
           end;
   PathType = ^Item;
   Item = record
             Data : integer;
             Next : PathType;
          end;
   CostMatrix = array [1..28,1..28] of real;

var
   A : CostMatrix;
   List : VerticeType;
   LinkedPath : PathType;
   OBSTACLES, GSPACE, VERTICES, PATH : text;






procedure BuildGSpaceObstacles(var List : VerticeType);
{________________________________________________________________

    Author  :  C. H. Chung

    Version :  2.3

    Date    :  November 29, 1988

   ________________________________________________________________

    Procedure BuildGSpaceObstacles(List).
               .  To build the Grown Space Obstacles
               .  INPUT  FILE : OBSTACLES, (GSPACE)
               .  OUTPUT FILE : output, GSPACE
               .  output of this procedure : List


    This program will build the Grown Space Obstacles.

```

```
     hh : the horizontal length of the object

     vv : the vertical length of the object

     rr : the sliced angle for rotational Grown Space Obstacles (radian
    ___________________________________________________________________

const
   Pi = 3.141592;    {Radian}
var
   Object : VerticeType;
   ObjectA, ObjectB, ObjectC : VerticeType;
   hh, vv, rr : real;
   From, To : PointType;





     procedure PrintVertice(List : VerticeType);
     {____________________________________________________________
        This procedure will print the Linked List for INPUT.
        List contains the Start, Goal, and Obstacles.
      ____________________________________________________________
     var
        Current : VerticeType;
     begin
        Current := List;
        writeln;
        writeln(Current^.Node.x :10:3, Current^.Node.y :10:3);
        Current := Current^.Next;
        while (Current<> nil)
           do begin
                 writeln(Current^.Node.x :10:3, Current^.Node.y :10:3)
                 Current := Current^.Next;
              end;
        writeln;
     end;






     procedure CreateObject(var Object : VerticeType);
     {____________________________________________________________
        This procedure creates the object from the input file
        by the linked list.
      ____________________________________________________________
```

```
var
   Current : VerticeType;
begin
   Object := nil;
   if not eof(OBSTACLES)
      then begin
              new(Object);
              readln(OBSTACLES, Object^.Node.x,
                                Object^.Node.y);
              Object^.Next := nil;
              Current := Object;
              while not eof(OBSTACLES)
                 do begin
                       new(Current^.Next);
                       Current := Current^.Next;
                       readln(OBSTACLES, Current^.Node.x,
                                         Current^.Node.y);
                       Current^.Next := nil
                    end
           end
end;




procedure CreateList(var List : VerticeType; From, To : PointType)
{______________________________________________________________
   This procedure creates the object from the input file
   by the linked list.
_______________________________________________________________}
var
   Current : VerticeType;
   Flag : boolean;
begin
   List := nil;
   if not eof(GSPACE)
      then begin
              new(List);
              List^.Node := From;
              List^.Next := nil;
              Current := List;

              new(Current^.Next);
              Current := Current^.Next;
              Current^.Node := From;
              Current^.Next := nil;

              Flag := true;
```

```
                    while Flag and (not eof(GSPACE))
                       do begin
                           new(Current^.Next);
                           Current := Current^.Next;
                           readln(GSPACE, Current^.Node.x,
                                          Current^.Node.y);
                           Current^.Next := nil;
                           if ((Current^.Node.x = 7.0) and   {* bad  *}
                              (Current^.Node.y = 1.0))       {STOPPING}
                             then Flag := false;             {* CASE *}
                         end;

                    new(Current^.Next);
                    Current := Current^.Next;
                    Current^.Node := To;
                    Current^.Next := nil;

                    new(Current^.Next);
                    Current := Current^.Next;
                    Current^.Node := To;
                    Current^.Next := nil;
                  end;
         end;


procedure GrownObject(var Object, Grown : VerticeType;
                          hh, vv,  rr : real);
{______________________________________________________________
    This procedure builds the Grown Space Obstacles.

                                    where h : horizontal length
                                          v : vertical length

    0 < q < Pi/2

        a1 = (A1x,A1y)  + h(-cos(q),-sin(q))
        a2 = (A1x,A1y)
        a3 = (A2x,A2y)
        a4 = (A2x,A2y)  + v(sin(q),-cos(q))
        a5 = (A3x,A3y)  + v(sin(q),-cos(q))
        a6 = (a5x,a5y)  + h(-cos(q),-sin(q))
        a8 = (A4x,A4y)  + h(-cos(q),-sin(q))
        a7 = (a8x,a8y)  + v(sin(q),-cos(q))

    Pi/2 < q < Pi
```

```
                q = q - Pi/2
                temp = h        (to swap h and v)
                h = v
                v = temp

        q = 0
                Delete a2, a4, a6, a8.

        q = Pi/2
                Swap h and v.
                Delete a2, a4, a6, a8.
```

```
var
   Current, Head : VerticeType;
begin
   Current := nil;
   new(Current);
   Current^.Node.x := Object^.Node.x - hh * cos(rr);
   Current^.Node.y := Object^.Node.y - hh * sin(rr);
   Current^.Next := nil;

   Head := Object;
   Grown := Current;

   new(Current^.Next);
   Current := Current^.Next;
   Current^.Node.x := Object^.Node.x;
   Current^.Node.y := Object^.Node.y;
   Current^.Next := nil;

   Object := Object^.Next;
   new(Current^.Next);
   Current := Current^.Next;
   Current^.Node.x := Object^.Node.x;
   Current^.Node.y := Object^.Node.y;
   Current^.Next := nil;

   new(Current^.Next);
   Current := Current^.Next;
   Current^.Node.x := Object^.Node.x + vv * sin(rr);
   Current^.Node.y := Object^.Node.y - vv * cos(rr);
   Current^.Next := nil;

   Object := Object^.Next;
   new(Current^.Next);
   Current := Current^.Next;
   Current^.Node.x := Object^.Node.x + vv * sin(rr);
   Current^.Node.y := Object^.Node.y - vv * cos(rr);
   Current^.Next := nil;
```

```
      new(Current^.Next);
      Current := Current^.Next;
      Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
      Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
      Current^.Next := nil;

      Object := Object^.Next;
      new(Current^.Next);
      Current := Current^.Next;
      Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr
      Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr)
      Current^.Next := nil;

      new(Current^.Next);
      Current := Current^.Next;
      Current^.Node.x := Object^.Node.x - hh * cos(rr);
      Current^.Node.y := Object^.Node.y - hh * sin(rr);
      Current^.Next := nil;

      Object := Head;
   end;




   procedure RotationalGrowth(ObjectA, ObjectB, ObjectC : VerticeType
                                 rr, hh, vv : real);
   {_______________________________________________________________
      This procedure will print the Grown Space Obstaces, considering
                                          the rotational effect.
   ________________________________________________________________}
   var
      GrownA1, GrownA2,
      GrownB1, GrownB2,
      GrownC1, GrownC2 : VerticeType;

   begin
      GrownObject(ObjectA, GrownA1, hh, vv, rr*0);
      GrownObject(ObjectB, GrownB1, hh, vv, rr*0);
      GrownObject(ObjectC, GrownC1, hh, vv, rr*0);
      GrownObject(ObjectA, GrownA2, vv, hh, rr*2 - Pi/2);
      GrownObject(ObjectB, GrownB2, vv, hh, rr*2 - Pi/2);
      GrownObject(ObjectC, GrownC2, vv, hh, rr*2 - Pi/2);

      while (GrownA1 <> nil) do
         begin
```

```
                writeln(GSPACE, GrownA1^.Node.x :10:4,
                                GrownA1^.Node.y :10:4);
                writeln(GSPACE, GrownA2^.Node.x :10:4,
                                GrownA2^.Node.y :10:4);
                writeln(GSPACE, GrownB1^.Node.x :10:4,
                                GrownB1^.Node.y :10:4);
                writeln(GSPACE, GrownB2^.Node.x :10:4,
                                GrownB2^.Node.y :10:4);
                writeln(GSPACE, GrownC1^.Node.x :10:4,
                                GrownC1^.Node.y :10:4);
                writeln(GSPACE, GrownC2^.Node.x :10:4,
                                GrownC2^.Node.y :10:4);

                GrownA1 := GrownA1^.Next^.Next;
                GrownA2 := GrownA2^.Next^.Next;
                GrownB1 := GrownB1^.Next^.Next;
                GrownB2 := GrownB2^.Next^.Next;
                GrownC1 := GrownC1^.Next^.Next;
                GrownC2 := GrownC2^.Next^.Next;
                writeln(GSPACE);
            end;
    end;  { of Procedure RotationalGrowth }





    procedure Partition(var Object, ObjectA, ObjectB,
                                            ObjectC : VerticeType);
    {_________________________________________________________________
       This procedure will partition the whole Object into 3 small
       objects (ObjectA, ObjectB, ObjectC).
    _________________________________________________________________}
    var
       i : integer;
       Current, CurrentA, CurrentB, CurrentC : VerticeType;
    begin
       Current := Object;

       ObjectA := nil;
       new(ObjectA);
       ObjectA^.Node := Current^.Node;
       ObjectA^.Next := nil;
       CurrentA := ObjectA;
       for i := 1 to 3 do
           begin
              new(CurrentA^.Next);
              Current := Current^.Next;
```

```
               CurrentA := CurrentA^.Next;
               CurrentA^.Node := Current^.Node;
               CurrentA^.Next := nil;
            end;

      Current := Current^.Next;

      ObjectB := nil;
      new(ObjectB);
      ObjectB^.Node := Current^.Node;
      ObjectB^.Next := nil;
      CurrentB := ObjectB;
      for i := 1 to 3 do
          begin
             new(CurrentB^.Next);
             Current := Current^.Next;
             CurrentB := CurrentB^.Next;
             CurrentB^.Node := Current^.Node;
             CurrentB^.Next := nil;
          end;

      Current := Current^.Next;

      ObjectC := nil;
      new(ObjectC);
      ObjectC^.Node := Current^.Node;
      ObjectC^.Next := nil;
      CurrentC := ObjectC;
      for i := 1 to 3 do
          begin
             new(CurrentC^.Next);
             Current := Current^.Next;
             CurrentC := CurrentC^.Next;
             CurrentC^.Node := Current^.Node;
             CurrentC^.Next := nil;
          end;
   end;  { of procedure Partition }



   begin   {_______ BuildGSpaceObstacles _______}

      reset(OBSTACLES);
      rewrite(GSPACE);

      rr := Pi / 4;
      hh := 2;
      vv := 1;

```

```
        CreateObject(Object);
        Partition(Object, ObjectA, ObjectB, ObjectC);
        RotationalGrowth(ObjectA, ObjectB, ObjectC, rr, hh, vv);

        reset(GSPACE); {***************************************}

        From.x := 5.0;              { From, To }
        From.y := 7.0;

        To.x := 13.0;   {***************************************}
        To.y := 15.0;

        CreateList(List, From, To);

        writeln('  ___  Linked List for INPUT ___  ');
        PrintVertice(List);
        writeln;
        writeln;
        writeln(' INPUT  FILE : OBSTACLES');
        writeln;
        writeln(' OUTPUT FILE : GSPACE   (for the Grown Space Obstacles
        writeln('               VERTICES (for the Visible Vertices)');
        writeln('               PATH     (for the shortest path)');
        writeln('               output   (for this display)');

     end;  { of Procedure BuildGSpaceObstacles }






procedure  BuildVGraph(var A : CostMatrix; var List : VerticeType);
{___________________________________________________________________

   Author   :  C. H. Chung

   Version :  2.0

   Date     :  November 17, 1988

  __________________________________________________________________

   This program will build the Visibility Graph.

   BuildVGraph(A, List)
        .  To build the VGraph
        .  The first part of this procedure mainly consists of
           the Interference Checking, i.e. the Visible Vertices,
```

```
            and the second part of this procedure mainly consists
            of the VGraph Construction.

    Pay a special attentation on the data structure of List.
    List consists of S1, S2,
                       A1(1), A2(1), B1(1), B2(1), C1(1), C2(1),
                       A1(2), A2(2), B1(2), B2(2), C1(2), C2(2),
                       A1(3), A2(3), B1(3), B2(3), C1(3), C2(3),
                       A1(4), A2(4), B1(4), B2(4), C1(4), C2(4),
                       G1, G2.
  ________________________________________________________________}

var
   From, To : VerticeType;
   i, j, n : integer;






     {________________________________________________________________
      This procedure is very useful in printing a Linked List
      to debug the procedure of BuildVGraph.  So, this procedure
      will remain in the main programming sheet for the future
      debugging.


     procedure PrintVertice(List : VerticeType);
     ________________________________________________________________
       This procedure will print the Linked List of the shortest path
     ________________________________________________________________
     var
        Current : VerticeType;
     begin
        Current := List;
        writeln(VERTICES,Current^.Node.x :10:3,
                              Current^.Node.y :10:3);
        Current := Current^.Next;
        while (Current<> nil)
           do begin
                 writeln(VERTICES,Current^.Node.x :10:3,
                                     Current^.Node.y :10:3);
                 Current := Current^.Next;
              end;
        writeln(VERTICES);
     end;
     ________________________________________________________________
```

```
procedure PrintAMatrix(A : CostMatrix);
{______________________________________________________________________________
   This procedure will print a Matrix.
________________________________________________________________________________}
var
   i, j : integer;

begin
   for i := 1 to 28 do
     begin
        writeln(VERTICES);
        for j := 1 to 28 do
            if A[i,j] < 999
               then writeln(VERTICES,'  A[', i :2:0,',',
                                  j :2:0,'] =', A[i,j] :10:4);
     end;
end;






procedure LineEquation(From, To : PointType; var a, b : real);
 {_____________________________________________________________________________
   This procedure will generate a line equation through two points
 ______________________________________________________________________________


         Y2 - Y1
   Y := ---------(X - X1) + Y1
         X2 - X1

         Y2 - Y1       X2*Y1 - Y2*X1
      := ---------(X) + ---------------
         X2 - X1          X2 - X1

      := a*X + b
 ______________________________________________________________________________}

begin
   a := (To.y - From.y) / (To.x - From.x);
   b := (To.x * From.y - To.y * From.x) / (To.x - From.x);
end;
```

```
function max(a, b : real): real;
{________________________________________________________________
   This function calulate the maximum.
 ________________________________________________________________
begin
   if a > b
      then max := a
      else max := b;
end;





function min(a, b : real): real;
{________________________________________________________________
    This function will calculate the minimum.
 ________________________________________________________________
begin
   if a > b
      then min := b
      else min := a;
end;







function FinalCheck(a, b, X1, X2, X3, X4,
                              Y1, Y2, Y3, Y4 : real): boolean;
{________________________________________________________________
   This function will find the Interference in the normal case.
 ________________________________________________________________

var
   X, Y,
   Xmin, Ymin,
   Xmax, Ymax : real;

begin
```

```
   Xmin := max(X1, X3);
   Xmax := min(X2, X4);
   Ymin := max(Y1, min(Y3, Y4));
   Ymax := min(Y2, max(Y3, Y4));

   Y := Y2;
   X := (Y - b) / a;
   if (X > Xmin) and (X < Xmax)
      then FinalCheck := true
      else begin
              X := X2;
              Y := a * X + b;
              if (Y > Ymin) and (Y < Ymax)
                 then FinalCheck := true
                 else begin
                         Y := Y1;
                         X := (Y - b) / a;
                         if (X > Xmin) and (X < Xmax)
                            then FinalCheck := true
                            else begin
                                    X := X1;
                                    Y := a * X + b;
                                    if (Y > Ymin) and (Y < Ymax)
                                       then FinalCheck := true
                                       else FinalCheck := false;
                                 end;
                      end;
           end;
end;





function DetectInterference(Object : VerticeType;
                            From, To : PointType) : boolean;
{______________________________________________________________
    This function will classify the Interference.
 ______________________________________________________________

var
   a, b,
   tempX, tempY,
   X1, X2, X3, X4,
   Y1, Y2, Y3, Y4 : real;
   Current : VerticeType;

begin
```

```
        Current := Object;

        {1.  Swap From and To by the X position,
             in order to set From to the left of To.}
        if From.x > To.x
           then begin
                    tempX := From.x;
                    tempY := From.y;
                    From.x := To.x;
                    From.y := To.y;
                    To.x := tempX;
                    To.y := tempY;
                end;

        {2.  Define X1, X2, X3, X4, Y1, Y2, Y3, Y4.}
        X1 := Current^.Node.x;
        Y2 := Current^.Node.y;
        Current := Current^.Next;
        X2 := Current^.Node.x;
        Current := Current^.Next;
        Y1 := Current^.Node.y;
        Current := Object;
        X3 := From.x;
        Y3 := From.y;
        X4 := To.x;
        Y4 := To.y;

        {3.  Find the x, y boundary of the object.
             However, Step 2 implies Step 3.}

{4.  Find the line equation throught From and To and its boundary.}
if (X3 = X4) and (Y3 = Y4)
   then DetectInterference := false
   else if (X3 = X4)
           then if (X3 > X1) and (X3 < X2)
                   then if (min(Y3,Y4) > Y2) or (max(Y3,Y4) < Y1)
                           then DetectInterference := false
                           else DetectInterference := true
                   else DetectInterference := false
           else if (Y3 = Y4)
                   then if (Y3 > Y1) and (Y3 < Y2)
                           then if (min(X3,X4) > X2) or
                                   (max(X3,X4) < X1)
                                   then DetectInterference := false
                                   else DetectInterference := true
                           else DetectInterference := false
                   else begin
                           LineEquation(From, To, a, b);

{5. & 6. is implied in DetectInterference. }
```

```
                                  if (min(Y3,Y4) >= Y2) or
                                     (max(Y3,Y4) <= Y1) or
                                     (min(X3,X4) >= X2) or
                                     (max(X3,X4) <= X1)
                                     then DetectInterference := false
                                     else DetectInterference :=
                                                 FinalCheck(a,b,X1,X2,
                                                    X3,X4,Y1,Y2,Y3,Y4)
                               end;
       end;




       {_______________________________________________________________
        This procedure is very useful in checking the Interference
        between the line and obstacles, and if there is any
        interference, then this procedure will print out the
        information on it.  However, this procedure will remain
        in the main programming sheet for the future debugging.


       procedure PrintInformationOnInterference(Object : VerticeType;
                                               From, To : PointType;
                                               First, Second : char);
       ________________________________________________________________
          This procedure will print the information on Interference.
       ________________________________________________________________
       var
          Current : VerticeType;
       begin
          Current := Object;
          if DetectInterference(Current, From, To)
             then begin
                     writeln(VERTICES,'  in Object ', First, Second);
                     PrintVertice(Current);
                  end;
       end;
       ________________________________________________________________}






       function Interference(List : VerticeType; From, To : PointType;
```

```
                         COUNT : integer): boolean;
{
   This function will find the Interference with the rotational
   Grown Space Obstacles.
 
}

var
   Current,
   HeadA1, HeadA2,
   HeadB1, HeadB2,
   HeadC1, HeadC2,
   ObjectA1, ObjectA2,
   ObjectB1, ObjectB2,
   ObjectC1, ObjectC2 : VerticeType;

begin
   Current := List;
   Current := Current^.Next;          { Skip S1.}
   Current := Current^.Next;          { Skip S2.}

   ObjectA1 := nil;
   new(ObjectA1);
   ObjectA1^.Node := Current^.Node;
   ObjectA1^.Next := nil;
   Current := Current^.Next;

   ObjectA2 := nil;
   new(ObjectA2);
   ObjectA2^.Node := Current^.Node;
   ObjectA2^.Next := nil;
   Current := Current^.Next;

   ObjectB1 := nil;
   new(ObjectB1);
   ObjectB1^.Node := Current^.Node;
   ObjectB1^.Next := nil;
   Current := Current^.Next;

   ObjectB2 := nil;
   new(ObjectB2);
   ObjectB2^.Node := Current^.Node;
   ObjectB2^.Next := nil;
   Current := Current^.Next;

   ObjectC1 := nil;
   new(ObjectC1);
   ObjectC1^.Node := Current^.Node;
   ObjectC1^.Next := nil;
   Current := Current^.Next;
```

```
        ObjectC2 := nil;
        new(ObjectC2);
        ObjectC2^.Node := Current^.Node;
        ObjectC2^.Next := nil;
        Current := Current^.Next;

        HeadA1 := ObjectA1;
        HeadA2 := ObjectA2;
        HeadB1 := ObjectB1;
        HeadB2 := ObjectB2;
        HeadC1 := ObjectC1;
        HeadC2 := ObjectC2;

        while Current^.Next^.Next <> nil      { Skip G1, G2.}
           do begin
                new(ObjectA1^.Next);
                ObjectA1 := ObjectA1^.Next;
                ObjectA1^.Node := Current^.Node;
                ObjectA1^.Next := nil;
                Current := Current^.Next;

                new(ObjectA2^.Next);
                ObjectA2 := ObjectA2^.Next;
                ObjectA2^.Node := Current^.Node;
                ObjectA2^.Next := nil;
                Current := Current^.Next;

                new(ObjectB1^.Next);
                ObjectB1 := ObjectB1^.Next;
                ObjectB1^.Node := Current^.Node;
                ObjectB1^.Next := nil;
                Current := Current^.Next;

                new(ObjectB2^.Next);
                ObjectB2 := ObjectB2^.Next;
                ObjectB2^.Node := Current^.Node;
                ObjectB2^.Next := nil;
                Current := Current^.Next;

                new(ObjectC1^.Next);
                ObjectC1 := ObjectC1^.Next;
                ObjectC1^.Node := Current^.Node;
                ObjectC1^.Next := nil;
                Current := Current^.Next;

                new(ObjectC2^.Next);
                ObjectC2 := ObjectC2^.Next;
                ObjectC2^.Node := Current^.Node;
                ObjectC2^.Next := nil;
                Current := Current^.Next;
```

```
            end;

      ObjectA1 := HeadA1;
      ObjectA2 := HeadA2;
      ObjectB1 := HeadB1;
      ObjectB2 := HeadB2;
      ObjectC1 := HeadC1;
      ObjectC2 := HeadC2;

      if COUNT = COUNT div 2 * 2
         then if DetectInterference(ObjectA2, From, To) or
                 DetectInterference(ObjectB2, From, To) or
                 DetectInterference(ObjectC2, From, To)
                 then Interference := true
                 else Interference := false
         else if DetectInterference(ObjectA1, From, To) or
                 DetectInterference(ObjectB1, From, To) or
                 DetectInterference(ObjectC1, From, To)
                 then Interference := true
                 else Interference := false;

   end;  { Of function Interference }






   function CrossDiagonal(Object : VerticeType;
                          From, To : PointType): boolean;
   {____________________________________________________________
      This function will determine whether two vertices are in
      the disgoanl of the same object.
   ____________________________________________________________
   var
      Current1, Current2, Current3, Current4 : VerticeType;
      tempX, tempY : real;
   begin
      Current1 := Object;
      Current2 := Current1^.Next;
      Current3 := Current2^.Next;
      Current4 := Current3^.Next;

      if From.x > To.x
         then begin
                 tempX := From.x;
                 tempY := From.y;
                 From.x := To.x;
```

```
                    From.y := To.y;
                    To.x := tempX;
                    To.y := tempY;
                 end;

         if (Current1^.Node.x = From.x) and
            (Current1^.Node.y = From.y) and
            (Current3^.Node.x = To.x) and
            (Current3^.Node.y = To.y)
            then CrossDiagonal := true
            else if (Current2^.Node.x = To.x) and
                    (Current2^.Node.y = To.y) and
                    (Current4^.Node.x = From.x) and
                    (Current4^.Node.y = From.y)
                    then CrossDiagonal := true
                    else CrossDiagonal := false;
      end;   { Of function CrossDiagonal  }







      function CrossVertices(List : VerticeType;
                             From, To : PointType) : boolean;
      {________________________________________________________________
         This function will find the Interference with the rotational
         Grown Space Obstacles.
       ________________________________________________________________

      var
         Current,
         HeadA1, HeadA2,
         HeadB1, HeadB2,
         HeadC1, HeadC2,
         ObjectA1, ObjectA2,
         ObjectB1, ObjectB2,
         ObjectC1, ObjectC2 : VerticeType;

      begin
         Current := List;
         Current := Current^.Next;           { Skip S1.}
         Current := Current^.Next;           { Skip S2.}

         ObjectA1 := nil;
         new(ObjectA1);
         ObjectA1^.Node := Current^.Node;
         ObjectA1^.Next := nil;
         Current := Current^.Next;
```

```
        ObjectA2 := nil;
        new(ObjectA2);
        ObjectA2^.Node := Current^.Node;
        ObjectA2^.Next := nil;
        Current := Current^.Next;

        ObjectB1 := nil;
        new(ObjectB1);
        ObjectB1^.Node := Current^.Node;
        ObjectB1^.Next := nil;
        Current := Current^.Next;

        ObjectB2 := nil;
        new(ObjectB2);
        ObjectB2^.Node := Current^.Node;
        ObjectB2^.Next := nil;
        Current := Current^.Next;

        ObjectC1 := nil;
        new(ObjectC1);
        ObjectC1^.Node := Current^.Node;
        ObjectC1^.Next := nil;
        Current := Current^.Next;

        ObjectC2 := nil;
        new(ObjectC2);
        ObjectC2^.Node := Current^.Node;
        ObjectC2^.Next := nil;
        Current := Current^.Next;

        HeadA1 := ObjectA1;
        HeadA2 := ObjectA2;
        HeadB1 := ObjectB1;
        HeadB2 := ObjectB2;
        HeadC1 := ObjectC1;
        HeadC2 := ObjectC2;

        while Current^.Next^.Next <> nil      { Skip G1, G2.}
            do begin
                new(ObjectA1^.Next);
                ObjectA1 := ObjectA1^.Next;
                ObjectA1^.Node := Current^.Node;
                ObjectA1^.Next := nil;
                Current := Current^.Next;

                new(ObjectA2^.Next);
                ObjectA2 := ObjectA2^.Next;
                ObjectA2^.Node := Current^.Node;
                ObjectA2^.Next := nil;
```

```
                    Current := Current^.Next;

                    new(ObjectB1^.Next);
                    ObjectB1 := ObjectB1^.Next;
                    ObjectB1^.Node := Current^.Node;
                    ObjectB1^.Next := nil;
                    Current := Current^.Next;

                    new(ObjectB2^.Next);
                    ObjectB2 := ObjectB2^.Next;
                    ObjectB2^.Node := Current^.Node;
                    ObjectB2^.Next := nil;
                    Current := Current^.Next;

                    new(ObjectC1^.Next);
                    ObjectC1 := ObjectC1^.Next;
                    ObjectC1^.Node := Current^.Node;
                    ObjectC1^.Next := nil;
                    Current := Current^.Next;

                    new(ObjectC2^.Next);
                    ObjectC2 := ObjectC2^.Next;
                    ObjectC2^.Node := Current^.Node;
                    ObjectC2^.Next := nil;
                    Current := Current^.Next;

                end;

        ObjectA1 := HeadA1;
        ObjectA2 := HeadA2;
        ObjectB1 := HeadB1;
        ObjectB2 := HeadB2;
        ObjectC1 := HeadC1;
        ObjectC2 := HeadC2;

        if CrossDiagonal(ObjectA1, From, To) or
           CrossDiagonal(ObjectA2, From, To) or
           CrossDiagonal(ObjectB1, From, To) or
           CrossDiagonal(ObjectB2, From, To) or
           CrossDiagonal(ObjectC1, From, To) or
           CrossDiagonal(ObjectC2, From, To)
           then CrossVertices := true
           else CrossVertices := false;
    end;  {  Of Function CrossVertices }
```

```
function GrownDeadNode(Object : VerticeType;
                       Data: PointType): boolean;
{_______________________________________________________________________
   This function will classify the Dead node.
 ______________________________________________________________________}

var
   X1, X2, X3,
   Y1, Y2, Y3 : real;
   Current : VerticeType;

begin
   Current := Object;

   X1 := Current^.Node.x;
   Y2 := Current^.Node.y;
   Current := Current^.Next;
   X2 := Current^.Node.x;
   Current := Current^.Next;
   Y1 := Current^.Node.y;
   Current := Object;
   X3 := Data.x;
   Y3 := Data.y;
   if (X3 > X1) and (X3 < X2) and (Y3 > Y1) and (Y3 < Y2)
      then GrownDeadNode := true
      else GrownDeadNode := false;
end;





function DeadNode(List : VerticeType; Data : PointType;
                  i : integer): boolean;
{_______________________________________________________________________
   This function will find the Dead Node with the rotational
   Grown Space Obstacles.
 ______________________________________________________________________}

var
   Current,
   HeadA1, HeadA2,
   HeadB1, HeadB2,
   HeadC1, HeadC2,
   ObjectA1, ObjectA2,
   ObjectB1, ObjectB2,
   ObjectC1, ObjectC2 : VerticeType;

```

```
begin
   Current := List;
   Current := Current^.Next;         { Skip S1.}
   Current := Current^.Next;         { Skip S2.}

   ObjectA1 := nil;
   new(ObjectA1);
   ObjectA1^.Node := Current^.Node;
   ObjectA1^.Next := nil;
   Current := Current^.Next;

   ObjectA2 := nil;
   new(ObjectA2);
   ObjectA2^.Node := Current^.Node;
   ObjectA2^.Next := nil;
   Current := Current^.Next;

   ObjectB1 := nil;
   new(ObjectB1);
   ObjectB1^.Node := Current^.Node;
   ObjectB1^.Next := nil;
   Current := Current^.Next;

   ObjectB2 := nil;
   new(ObjectB2);
   ObjectB2^.Node := Current^.Node;
   ObjectB2^.Next := nil;
   Current := Current^.Next;

   ObjectC1 := nil;
   new(ObjectC1);
   ObjectC1^.Node := Current^.Node;
   ObjectC1^.Next := nil;
   Current := Current^.Next;

   ObjectC2 := nil;
   new(ObjectC2);
   ObjectC2^.Node := Current^.Node;
   ObjectC2^.Next := nil;
   Current := Current^.Next;

   HeadA1 := ObjectA1;
   HeadA2 := ObjectA2;
   HeadB1 := ObjectB1;
   HeadB2 := ObjectB2;
   HeadC1 := ObjectC1;
   HeadC2 := ObjectC2;

   while Current^.Next^.Next <> nil      { Skip G1, G2.}
      do begin
```

```
                new(ObjectA1^.Next);
                ObjectA1 := ObjectA1^.Next;
                ObjectA1^.Node := Current^.Node;
                ObjectA1^.Next := nil;
                Current := Current^.Next;

                new(ObjectA2^.Next);
                ObjectA2 := ObjectA2^.Next;
                ObjectA2^.Node := Current^.Node;
                ObjectA2^.Next := nil;
                Current := Current^.Next;

                new(ObjectB1^.Next);
                ObjectB1 := ObjectB1^.Next;
                ObjectB1^.Node := Current^.Node;
                ObjectB1^.Next := nil;
                Current := Current^.Next;

                new(ObjectB2^.Next);
                ObjectB2 := ObjectB2^.Next;
                ObjectB2^.Node := Current^.Node;
                ObjectB2^.Next := nil;
                Current := Current^.Next;

                new(ObjectC1^.Next);
                ObjectC1 := ObjectC1^.Next;
                ObjectC1^.Node := Current^.Node;
                ObjectC1^.Next := nil;
                Current := Current^.Next;
                new(ObjectC2^.Next);
                ObjectC2 := ObjectC2^.Next;
                ObjectC2^.Node := Current^.Node;
                ObjectC2^.Next := nil;
                Current := Current^.Next;

             end;

        ObjectA1 := HeadA1;
        ObjectA2 := HeadA2;
        ObjectB1 := HeadB1;
        ObjectB2 := HeadB2;
        ObjectC1 := HeadC1;
        ObjectC2 := HeadC2;

        if (i = i div 2 * 2)
           then if GrownDeadNode(ObjectA2, Data) or
                   GrownDeadNode(ObjectB2, Data) or
                   GrownDeadNode(ObjectC2, Data)
                   then DeadNode := true
                   else DeadNode := false
```


a-2

```
                  else if GrownDeadNode(ObjectA1, Data) or
                          GrownDeadNode(ObjectB1, Data) or
                          GrownDeadNode(ObjectC1, Data)
                          then DeadNode := true
                          else DeadNode := false;
       end;







begin                                                      { BuildVGraph }
   rewrite(VERTICES);

   n := 28;    { Dimension of Cost[n,n] }
   for i := 1 to n do
   for j := 1 to n do
      A[i,j] := 9999.99;    { 9999.99 means the infinity }

   From := List;
   To := List;
   i := 0;
   j := 0;

   while From <> nil do
      begin
         i := i +1;
         if DeadNode(List, From^.Node, i)
            then               { nothing }
            else begin
                    To := List;
                    j := 0;
                    while To <> nil do
                       begin
                          j := j + 1;
                          if (From = To) or
                             CrossVertices(List, From^.Node, To^.Node)
                             DeadNode(List, From^.Node, j) or
                             DeadNode(List, To^.Node, j) or
                             Interference(List, From^.Node, To^.Node,
                             then                   { nothing }
                             else A[i,j] := sqrt(
                                                 (From^.Node.x - To^.Node.x
                                                 (From^.Node.x - To^.Node.x
                                                 (From^.Node.y - To^.Node.y
                                                 (From^.Node.y - To^.Node.y
                          To := To^.Next;
                       end;
```

```
                end;
            From := From^.Next;
         end;

      PrintAMatrix(A);

end;  { Of Procedure BuildVGraph }

procedure SearchVGraph(A : CostMatrix; var LinkedPath : PathType);
{______________________________________________________________________

     Author  :  C. H. Chung

     Version :  2.7

     Date    :  November 7, 1988
 ______________________________________________________________________

     This program will calulate the shortest path by Floyd Algorithm.

     SearchVGraph(A, LinkedPath)
           .  To find the shortest path of the VGraph
           .  LinkedPath holds the information of the shortest path
              by the VGraph Algorithm.
           .  INPUT  FILE : ____
           .  OUTPUT FILE : PATH
           .  input  of this procedure : A
           .  output of this procedure : LinkedPath

     LinkedPath : PathType;  *** output of procedure SearchVGraph ***
                         /*  Linked LinkedPath for the shortest path */
     A : CostMatrix;     /*  A : Cost matrix for Floyd Algorithm     */
     P : PathMatrix;     /*  P : Path Matrix for Floyd Algorithm     */
     n : integer;        /*  Dimension of Cost (Path) Matrix         */
     Cost : real;        /*  Cost of the shortest path

 ______________________________________________________________________}

type
    PathMatrix = array [1..28,1..28] of integer;
```

```
var
   P : PathMatrix;         { Path Matrix for the Floyd Algorithm  }
   n : integer;            { Dimension of the Cost (Path) Matrix  }
   Cost : real;            { Cost of the shortest path            }







      procedure InitializePath(var P : PathMatrix;
                               var LinkedPath : PathType;
                               var n : integer);
      {________________________________________________________________
        This procedure will initialize the Cost Matrix and Path Matrix.
        The Path Matrix are automatically set to zero.
        The Cost Matrix should be defined by User.
        The Start node and Goal node should be defined by User

        9999 means the infinitive.
       ________________________________________________________________
      var
         i, j : integer;
         Start, Goal : integer;

      begin
         n := 28;
         Start := 1;                    {**********************************
         Goal := 27;                                     { Start, Goal }
                                        {**********************************
         for i := 1 to n do
         for j := 1 to n do
                P[i,j] := 0;

         LinkedPath := nil;
         new(LinkedPath);
         LinkedPath^.Data := Start;
         LinkedPath^.Next := nil;
         new(LinkedPath^.Next);
         LinkedPath^.Next^.Data := Goal;
         LinkedPath^.Next^.Next := nil
      end;
```

```
procedure PrintPath(LinkedPath : PathType);
{
  This procedure will print the Linked LinkedPath of the shortest
  path.
}
var
   Current : PathType;
begin
   write(PATH, ' Path represented by internal nodes = ');
   Current := LinkedPath;
   write(PATH, Current^.Data :7);
   Current := Current^.Next;
   while (Current<> nil)
      do begin
            write(PATH, Current^.Data :7);
            Current := Current^.Next;
         end;
   writeln(PATH);
end;






procedure WriteVertice(i : integer);
{
   This procedure prints vertice.
}
begin
   case i of
      1 : write(PATH, ' START in  0` sliced');
      2 : write(PATH, ' START in 90` sliced');
      3 : write(PATH, '    A1 in  0` sliced');
      9 : write(PATH, '    A2 in  0` sliced');
     15 : write(PATH, '    A3 in  0` sliced');
     21 : write(PATH, '    A4 in  0` sliced');
      5 : write(PATH, '    B1 in  0` sliced');
     11 : write(PATH, '    B2 in  0` sliced');
     17 : write(PATH, '    B3 in  0` sliced');
     23 : write(PATH, '    B4 in  0` sliced');
      7 : write(PATH, '    C1 in  0` sliced');
     13 : write(PATH, '    C2 in  0` sliced');
     19 : write(PATH, '    C3 in  0` sliced');
     25 : write(PATH, '    C4 in  0` sliced');
      4 : write(PATH, '    A1 in 90` sliced');
     10 : write(PATH, '    A2 in 90` sliced');
     16 : write(PATH, '    A3 in 90` sliced');
     22 : write(PATH, '    A4 in 90` sliced');
```

```
          6 : write(PATH, '    B1 in 90` sliced');
         12 : write(PATH, '    B2 in 90` sliced');
         18 : write(PATH, '    B3 in 90` sliced');
         24 : write(PATH, '    B4 in 90` sliced');
          8 : write(PATH, '    C1 in 90` sliced');
         14 : write(PATH, '    C2 in 90` sliced');
         20 : write(PATH, '    C3 in 90` sliced');
         26 : write(PATH, '    C4 in 90` sliced');
         27 : write(PATH, '  GOAL in  0` sliced');
         28 : write(PATH, '  GOAL in 90` sliced');
      end;
   end;








   procedure PrintPrettyLinkedPath(A: CostMatrix;
                                   var LinkedPath : PathType);
   {____________________________________________________________________
     This procedure will print all nodes of the shortest path and
                                  rotation between any two nodes.
     PrintPath will print the Linked LinkedPath of the shortest
                                                            path.
   _____________________________________________________________________}
   var
      Current : PathType;
      Flag : integer;
      first, second : integer;
      i, j : integer;
   begin
      PrintPath(LinkedPath);
      write(PATH, '__________________________________________');
      writeln(PATH, '_______________________________________');
      writeln(PATH);
      writeln(PATH);
      write(PATH, '       From                                 To');
      writeln(PATH, '                  Cost  Rotation ');
      Current := LinkedPath;
      first := Current^.Data mod 2;              { 2 is related with
                                                   the sliced-angle.}
      second := Current^.Next^.Data mod 2;
      Flag := first - second;
      i := Current^.Data;
      j := Current^.Next^.Data;
      writeln(PATH);
      WriteVertice(i);
      write(PATH, '  -------> ');
```

```
      WriteVertice(j);
      write(PATH, '     ',A[i,j] :7:3);
      case Flag of
        -1 : writeln(PATH, '    <-90`>');
         0 : writeln(PATH, '    <0`>');
         1 : writeln(PATH, '    <90`>');
      end;
      Current := Current^.Next;

      while (Current^.Next <> nil)
         do begin
               i := Current^.Data;
               j := Current^.Next^.Data;
               if (Current^.Next^.Next = nil) or
                  (Current^.Next^.Data > 26)
                  then j := Current^.Next^.Data;
               WriteVertice(i);
               write(PATH, '  -------> ');
               WriteVertice(j);
               write(PATH, '     ', A[i,Current^.Next^.Data] :7:3);
               first := Current^.Data mod 2;
               second := Current^.Next^.Data mod 2;
               Flag := first - second;
               case Flag of
                 -1 : writeln(PATH, '    <-90`>');
                  0 : writeln(PATH, '    <0`>');
                  1 : writeln(PATH, '    <90`>');
               end;
            Current := Current^.Next;
         end;
   end;






   procedure PrintCostAndPath(A : CostMatrix; LinkedPath : PathTyp
                                  Cost : real);
   {_______________________________________________________________
      This procedure will print the shortest path and its cost.
    _______________________________________________________________}
   var
      Current : PathType;
      Start, Goal : integer;

   begin
      Current := LinkedPath;
      Start := Current^.Data;
```

```
      while (Current^.Next <> nil)
         do Current := Current^.Next;
      Goal := Current^.Data;

      writeln(PATH);
      writeln(PATH);
      writeln(PATH);
      writeln(PATH);
      write(PATH, '_____________________________________________');
      writeln(PATH, '_____________________________________________________');
      writeln(PATH);
      write(PATH, ' The shortest path is calculated by');
      writeln(PATH, ' the Graph Search Algorithm.');
      writeln(PATH);
      writeln(PATH, '                  Start Node = ', Start :3,
                    '     Goal Node = ', Goal :3);
      writeln(PATH);
      write(PATH, '_______________________________________________');
      writeln(PATH, '_________________________________________________');
      writeln(PATH);
      PrintPrettyLinkedPath(A,LinkedPath);
      writeln(PATH);
      write(PATH, '_______________________________________________');
      writeln(PATH, '_________________________________________________
      writeln(PATH);
      writeln(PATH, '   The Total Cost =  ', Cost :6:3);
      write(PATH, '________________________________________________');
      writeln(PATH, '________________________________________________');
      writeln(PATH);
      writeln(PATH);
   end;





procedure FindPath(var P : PathMatrix; var LinkedPath : PathType
{______________________________________________________________________
    This procedure will find the shortest path from Path Matrix.
 ______________________________________________________________________
var
   i, j : integer;
   Current : PathType;

begin
   i := LinkedPath^.Data;
   j := LinkedPath^.Next^.Data;
   Current := nil;
```

```
          if P[i,j] = 0
             then
             else begin
                     new(Current);
                     Current^.Data := P[i,j];
                     Current^.Next := LinkedPath^.Next;
                     LinkedPath^.Next := Current;
                     FindPath(P,LinkedPath);
                     FindPath(P,LinkedPath^.Next);
                  end;
       end;






       procedure FindCost(A : CostMatrix; LinkedPath : PathType;
                          var Cost : real);
       {______________________________________________________________
           This procedure will find the cost of the shortest path
           from the Cost Matrix.
        ______________________________________________________________}
       var
          Current : PathType;
          i, j : integer;

       begin
          Current := LinkedPath;
          Cost := 0;

          while (Current^.Next <> nil)
             do begin
                   i := Current^.Data;
                   j := Current^.Next^.Data;
                   Cost := Cost + A[i,j];
                   Current := Current^.Next;
                end;
       end;







       procedure CalculateCostAndPath(var A : CostMatrix;
                                      var P : PathMatrix;
                                      n : integer);
```

```
{______________________________________________________________________
    This procedure will calculate the Cost Matrix and the Path
    Matrix by the Floyd Algorithm.

    _______________
    A : Cost Matrix
    P : Path Matrix

______________________________________________________________________}
var
   order : integer;
   i, j : integer;
   value : real;

begin
   order := 1;
   repeat
      for i := 1 to n do
      for j := 1 to n do
          if ((i <> order) and (j <> order))
             then begin
                     value := A[i,order] + A[order,j];
                     if (A[i,j] > value)
                        then begin
                               A[i,j] := value;
                               P[i,j] := order;
                             end;
                  end;
    order := order + 1;
   until  not (order <= n)
end;





begin   {______  Procedure SearchVGraph  ______}

   rewrite(PATH);

   InitializePath(P,LinkedPath,n);
   CalculateCostAndPath(A,P,n);
   FindPath(P,LinkedPath);
   FindCost(A,LinkedPath,Cost);
   PrintCostAndPath(A,LinkedPath,Cost);

end;    {______  Of procedure SearchVGraph  ______}
```

```
begin  {________  Main  ________}

   BuildGSpaceObstacles(List);
   BuildVGraph(A, List);
   SearchVGraph(A, LinkedPath);

end.   {______ of Main  ________}
```

Appendix B: I/O FIELS for the VGraph Algorithm

[OBSTACLES]

3.0000	18.0000
9.0000	18.0000
9.0000	10.0000
3.0000	10.0000
10.5000	13.0000
19.0000	13.0000
19.0000	9.0000
10.5000	9.0000
8.0000	7.5000
16.0000	7.5000
16.0000	3.0000
8.0000	3.0000

[GSPACE]

1.0000	18.0000
2.0000	18.0000
8.5000	13.0000
9.5000	13.0000
6.0000	7.5000
7.0000	7.5000
9.0000	18.0000
9.0000	18.0000
19.0000	13.0000
19.0000	13.0000
16.0000	7.5000
16.0000	7.5000
9.0000	9.0000
9.0000	8.0000
19.0000	8.0000
19.0000	7.0000
16.0000	2.0000
16.0000	1.0000
1.0000	9.0000
2.0000	8.0000
8.5000	8.0000
9.5000	7.0000
6.0000	2.0000
7.0000	1.0000

[VERTICES]

```
A[ 1, 2] =    0.0000
A[ 1, 7] =    1.1180
A[ 1, 8] =    2.0616
A[ 1,16] =    4.1231
A[ 1,21] =    4.4721
A[ 1,22] =    3.1623
A[ 1,25] =    5.0990
A[ 1,26] =    6.3246

A[ 2, 1] =    0.0000
A[ 2, 7] =    1.1180
A[ 2, 8] =    2.0616
A[ 2,16] =    4.1231
A[ 2,21] =    4.4721
A[ 2,22] =    3.1623
A[ 2,25] =    5.0990
A[ 2,26] =    6.3246

A[ 3, 4] =    1.0000
A[ 3, 9] =    8.0000
A[ 3,10] =    8.0000
A[ 3,21] =    9.0000
A[ 3,22] =   10.0499

A[ 4, 3] =    1.0000
A[ 4, 9] =    7.0000
A[ 4,10] =    7.0000
A[ 4,21] =    9.0554
A[ 4,22] =   10.0000


A[ 6, 9] =    5.0249
A[ 6,10] =    5.0249
A[ 6,11] =    9.5000
A[ 6,12] =    9.5000
A[ 6,16] =    5.0249
A[ 6,17] =   10.7355
A[ 6,27] =    4.0311
A[ 6,28] =    4.0311

A[ 7, 1] =    1.1180
A[ 7, 2] =    1.1180
A[ 7, 8] =    1.0000
A[ 7,13] =   10.0000
A[ 7,16] =    3.0414
A[ 7,17] =   13.0096
A[ 7,21] =    5.2202
A[ 7,22] =    4.0311
A[ 7,23] =    2.5495
```

```
A[ 7,25] =     5.5000
A[ 7,26] =     6.5765

A[ 8, 2] =     2.0616
A[ 8, 7] =     1.0000
A[ 8,13] =     9.0000
A[ 8,16] =     2.0616
A[ 8,17] =    12.0104
A[ 8,19] =    10.5475
A[ 8,21] =     6.1847
A[ 8,22] =     5.0249
A[ 8,23] =     1.5811
A[ 8,25] =     5.5902
A[ 8,26] =     6.5000

A[ 9, 3] =     8.0000
A[ 9, 4] =     7.0000
A[ 9, 6] =     5.0249
A[ 9,10] =     0.0000
A[ 9,11] =    11.1803
A[ 9,12] =    11.1803
A[ 9,16] =    10.0000
A[ 9,27] =     5.0000
A[ 9,28] =     5.0000

A[10, 3] =     8.0000
A[10, 4] =     7.0000
A[10, 6] =     5.0249
A[10, 9] =     0.0000
A[10,11] =    11.1803
A[10,12] =    11.1803
A[10,16] =    10.0000
A[10,27] =     5.0000
A[10,28] =     5.0000

A[11, 6] =     9.5000
A[11, 9] =    11.1803
A[11,10] =    11.1803
A[11,12] =     0.0000
A[11,17] =     5.0000
A[11,18] =     6.0000
A[11,27] =     6.3246
A[11,28] =     6.3246

A[12, 6] =     9.5000
A[12, 9] =    11.1803
A[12,10] =    11.1803
A[12,11] =     0.0000
A[12,17] =     5.0000
A[12,18] =     6.0000
A[12,27] =     6.3246
A[12,28] =     6.3246
```

```
A[13, 7]  =  10.0000
A[13,17]  =   3.0414
A[13,19]  =   5.5000
A[13,23]  =   7.5166


A[16, 2]  =   4.1231
A[16, 6]  =   5.0249
A[16, 7]  =   3.0414
A[16, 8]  =   2.0616
A[16,10]  =  10.0000
A[16,11]  =  11.1803
A[16,13]  =   7.0178
A[16,17]  =  10.0000
A[16,22]  =   7.0000
A[16,23]  =   0.5000

A[17, 6]  =  10.7355
A[17, 7]  =  13.0096
A[17,11]  =   5.0000
A[17,12]  =   5.0000
A[17,13]  =   3.0414
A[17,18]  =   1.0000
A[17,19]  =   6.7082
A[17,23]  =  10.5000

A[18,11]  =   6.0000
A[18,12]  =   6.0000
A[18,13]  =   3.0414
A[18,17]  =   1.0000
A[18,19]  =   5.8310
A[18,20]  =   6.7082

A[19,13]  =   5.5000
A[19,17]  =   6.7082
A[19,18]  =   5.8310
A[19,20]  =   1.0000
A[19,25]  =  10.0000

A[20,13]  =   6.5000
A[20,17]  =   7.6158
A[20,18]  =   6.7082
A[20,19]  =   1.0000
A[20,25]  =  10.0499
A[20,26]  =   9.0000

A[21, 1]  =   4.4721
A[21, 3]  =   9.0000
A[21, 4]  =   9.0554
A[21, 7]  =   5.2202
A[21,22]  =   1.4142
A[21,23]  =   7.5664
A[21,25]  =   8.6023
A[21,26]  =  10.0000
```

```
A[22, 1]  =   3.1623
A[22, 2]  =   3.1623
A[22, 4]  =  10.0000
A[22, 7]  =   4.0311
A[22, 8]  =   5.0249
A[22,13]  =  14.0089
A[22,16]  =   7.0000
A[22,17]  =  17.0000
A[22,21]  =   1.4142
A[22,23]  =   6.5000
A[22,25]  =   7.2111
A[22,26]  =   8.6023

A[23, 2]  =   3.6401
A[23, 7]  =   2.5495
A[23, 8]  =   1.5811
A[23,10]  =  10.0125
A[23,13]  =   7.5166
A[23,16]  =   0.5000
A[23,17]  =  10.5000
A[23,21]  =   7.5664
A[23,22]  =   6.5000

A[25, 1]  =   5.0990
A[25, 2]  =   5.0990
A[25, 7]  =   5.5000
A[25, 8]  =   5.5902
A[25,19]  =  10.0000
A[25,21]  =   8.6023
A[25,22]  =   7.2111
A[25,26]  =   1.4142

A[26, 2]  =   6.3246
A[26, 8]  =   6.5000
A[26,19]  =   9.0554
A[26,20]  =   9.0000
A[26,22]  =   8.6023
A[26,25]  =   1.4142

A[27, 6]  =   4.0311
A[27, 9]  =   5.0000
A[27,10]  =   5.0000
A[27,11]  =   6.3246
A[27,12]  =   6.3246
A[27,28]  =   0.0000

A[28, 6]  =   4.0311
A[28, 9]  =   5.0000
A[28,10]  =   5.0000
A[28,11]  =   6.3246
A[28,12]  =   6.3246
A[28,27]  =   0.0000
```

[PATH]

```
The shortest path is calculated by the Graph Search Algorithm.
              Start Node =    1     Goal Node =  27

Path represented by internal nodes =        1      16       6      27

         From                              To
                                                              Cost   Rotation
START in  0` sliced  -------->     A3 in 90` sliced         4.123   <90`>
   A3 in 90` sliced  -------->     B1 in 90` sliced         5.025   <0`>
   B1 in 90` sliced  -------->   GOAL in  0` sliced         4.031   <-90`>

 The Total Cost =  13.179
```

Appendix C: Simulation of the Rotational GSpace

```
program BuildGrownSpaceObstaclesWithRotation(OBSTACLES, ROTATION);
{______________________________________________________________

   Author  :  C. H. Chung

   Version :  3.5

   Date    :  November 17, 1988

 ______________________________________________________________

   BuildGSpaceWithRotation;

              .  To build the Grown Space Obstacles with rotation
              .  INPUT  FILE : OBSTACLES
              .  OUTPUT FILE : ROTATION


   This program will build the Grown Space Obstacles.
 ______________________________________________________________
type
   Point2D = record
                  x, y : real;
               end;
   Vertice2D = ^Node2D;
   Node2D = record
               Node : Point2D;
               Next : Vertice2D
            end;
var
   OBSTACLES, ROTATION : text;





     procedure BuildGSpaceWithRotation;
     {_________________________________________________________

        Author  :  C. H. Chung

        Version :  2.3

        Date    :  December 3, 1988

      _________________________________________________________

        BuildGSpaceWithRotation;

                 . To build the Grown Space Obstacles with rotation
```

```
                  . INPUT  FILE : OBSTACLES
                  . OUTPUT FILE : ROTATION


      This program will build the Grown Space Obstacles.

      hh : the horizontal length of the object

      vv : the vertical length of the object

      rr : the sliced angle for rotational Grown Space Obstacles.
    _______________________________________________________________}
   const
      Pi = 3.141592;     {Radian}
   var
      Object : Vertice2D;
      ObjectA, ObjectB, ObjectC : Vertice2D;
      hh, vv, rr : real;





   procedure Print2Dvertice(List : Vertice2D);
   {__________________________________________________
      This procedure will print the Linked List of the
      shortest path.
   ____________________________________________________}
   begin
      if List = nil
         then writeln(ROTATION)
         else begin
                 writeln(ROTATION, List^.Node.x :10:4,
                                   List^.Node.y :10:4);
                 Print2Dvertice(List^.Next)
              end
   end;






   procedure CreateObject(var Object : Vertice2D);
   {_______________________________________________________
      This procedure creates the object from the input file
      by the linked list.
    _______________________________________________________}
   var
      Current : Vertice2D;
```

```
      begin
         Object := nil;
         if not eof(OBSTACLES)
            then begin
                    new(Object);
                    readln(OBSTACLES, Object^.Node.x,
                                      Object^.Node.y);
                    Object^.Next := nil;
                    Current := Object;
                    while not eof(OBSTACLES)
                       do begin
                             new(Current^.Next);
                             Current := Current^.Next;
                             readln(OBSTACLES, Current^.Node.x,
                                               Current^.Node.y);
                             Current^.Next := nil
                          end
                 end
      end;






      procedure GrownObject(var Object, Grown : Vertice2D;
                                hh, vv,  rr : real);
      {____________________________________________________________
          This procedure builds the Grown Space Obstacles.

                                           where h : horizontal length
                                                 v : vertical length

          0 < q < Pi/2

               a1 = (A1x,A1y)  + h(-cos(q),-sin(q))
               a2 = (A1x,A1y)
               a3 = (A2x,A2y)
               a4 = (A2x,A2y)  + v(sin(q),-cos(q))
               a5 = (A3x,A3y)  + v(sin(q),-cos(q))
               a6 = (a5x,a5y)  + h(-cos(q),-sin(q))
               a8 = (A4x,A4y)  + h(-cos(q),-sin(q))
               a7 = (a8x,a8y)  + v(sin(q),-cos(q))

          Pi/2 < q < Pi

               q = q - Pi/2
               temp = h     (to swap h and v)
               h = v
```

```
                v = temp

         q = 0
              Delete a2, a4, a6, a8.

         q = Pi/2
              Swap h and v.
              Delete a2, a4, a6, a8.
  ____________________________________________________________________}
     var
        Current, Head : Vertice2D;
     begin
        Current := nil;
        new(Current);
        Current^.Node.x := Object^.Node.x - hh * cos(rr);
        Current^.Node.y := Object^.Node.y - hh * sin(rr);
        Current^.Next := nil;

        Head := Object;
        Grown := Current;

        new(Current^.Next);
        Current := Current^.Next;
        Current^.Node.x := Object^.Node.x;
        Current^.Node.y := Object^.Node.y;
        Current^.Next := nil;

        Object := Object^.Next;
        new(Current^.Next);
        Current := Current^.Next;
        Current^.Node.x := Object^.Node.x;
        Current^.Node.y := Object^.Node.y;
        Current^.Next := nil;

        new(Current^.Next);
        Current := Current^.Next;
        Current^.Node.x := Object^.Node.x + vv * sin(rr);
        Current^.Node.y := Object^.Node.y - vv * cos(rr);
        Current^.Next := nil;

        Object := Object^.Next;
        new(Current^.Next);
        Current := Current^.Next;
        Current^.Node.x := Object^.Node.x + vv * sin(rr);
        Current^.Node.y := Object^.Node.y - vv * cos(rr);
        Current^.Next := nil;

        new(Current^.Next);
        Current := Current^.Next;
        Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
```

```
    Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
    Current^.Next := nil;

    Object := Object^.Next;
    new(Current^.Next);
    Current := Current^.Next;
    Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
    Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
    Current^.Next := nil;

    new(Current^.Next);
    Current := Current^.Next;
    Current^.Node.x := Object^.Node.x - hh * cos(rr);
    Current^.Node.y := Object^.Node.y - hh * sin(rr);
    Current^.Next := nil;

    Object := Head;

    if (rr = 0) or (abs(rr - Pi/2) < 0.001)
       then begin
               Current := nil;
               new(Current);
               Current^.Node := Grown^.Node;
               Current^.Next := nil;

               Head := nil;
               Head := Current;
               while Grown^.Next^.Next <> nil
                  do begin
                        new(Current^.Next);
                        Current := Current^.Next;
                        Grown := Grown^.Next^.Next;
                        Current^.Node := Grown^.Node;
                        Current^.Next := nil;
                     end;
               Grown := Head;
            end;
end;




procedure RotationObjects(ObjectA, ObjectB, ObjectC : Vertice2D;
                          rr, hh, vv : real);
{_____________________________________________________________________
   This procedure will print the GSpace Obstacle with Rotation.
_____________________________________________________________________}
var
```

```
      GrownA, GrownB, GrownC : Vertice2D;
      i : integer;
      Angle : real;
   begin
      i := 0;
      Angle := rr * i;

      while (Angle >= 0) and (Angle < Pi) do
         begin
            writeln(ROTATION, Angle*180/Pi :5:1, '` Rotation');
            if Angle = 0
               then begin
                     GrownObject(ObjectA, GrownA, hh, vv, Angle);
                     GrownObject(ObjectB, GrownB, hh, vv, Angle);
                     GrownObject(ObjectC, GrownC, hh, vv, Angle);
                     Print2Dvertice(GrownA);
                     Print2Dvertice(GrownB);
                     Print2Dvertice(GrownC);
                  end;
            if (Angle > 0) and (Angle < Pi/2)
               then begin
                     GrownObject(ObjectA, GrownA, hh, [illegible]
                     GrownObject(ObjectB, GrownB, hh, vv, Angle);
                     GrownObject(ObjectC, GrownC, hh, vv, Angle);
                     Print2Dvertice(GrownA);
                     Print2Dvertice(GrownB);
                     Print2Dvertice(GrownC);
                  end;
             if abs(Angle - Pi/2) < 0.001   { because of Round Off }
               then begin
                     GrownObject(ObjectA,GrownA,vv,hh,Angle-Pi/2);
                     GrownObject(ObjectB,GrownB,vv,hh,Angle-Pi/2);
                     GrownObject(ObjectC,GrownC,vv,hh,Angle-Pi/2);
                     Print2Dvertice(GrownA);
                     Print2Dvertice(GrownB);
                     Print2Dvertice(GrownC);
                  end
               else if (Angle > Pi/2)
                     then begin
                     GrownObject(ObjectA,GrownA,vv,hh,Angle-Pi/2);
                     GrownObject(ObjectB,GrownB,vv,hh,Angle-Pi/2);
                     GrownObject(ObjectC,GrownC,vv,hh,Angle-Pi/2);
                     Print2Dvertice(GrownA);
                     Print2Dvertice(GrownB);
                     Print2Dvertice(GrownC);
                         end;
            i := i + 1;
            Angle := rr * i;
         end;
   end;
```

```
procedure Partition(var Object, ObjectA, ObjectB,
                              ObjectC : Vertice2D);
{________________________________________________________
   This procedure will partition the whole Object into
   3 objects.
_________________________________________________________}
var
   i : integer;
   Current, CurrentA, CurrentB, CurrentC : Vertice2D;
begin
   Current := Object;

   ObjectA := nil;
   new(ObjectA);
   ObjectA^.Node := Current^.Node;
   ObjectA^.Next := nil;
   CurrentA := ObjectA;
   for i := 1 to 3 do
       begin
          new(CurrentA^.Next);
          Current := Current^.Next;
          CurrentA := CurrentA^.Next;
          CurrentA^.Node := Current^.Node;
          CurrentA^.Next := nil;
       end;

   Current := Current^.Next;

   ObjectB := nil;
   new(ObjectB);
   ObjectB^.Node := Current^.Node;
   ObjectB^.Next := nil;
   CurrentB := ObjectB;
   for i := 1 to 3 do
       begin
          new(CurrentB^.Next);
          Current := Current^.Next;
          CurrentB := CurrentB^.Next;
          CurrentB^.Node := Current^.Node;
          CurrentB^.Next := nil;
       end;

   Current := Current^.Next;

   ObjectC := nil;
```

```
        new(ObjectC);
        ObjectC^.Node := Current^.Node;
        ObjectC^.Next := nil;
        CurrentC := ObjectC;
        for i := 1 to 3 do
            begin
               new(CurrentC^.Next);
               Current := Current^.Next;
               CurrentC := CurrentC^.Next;
               CurrentC^.Node := Current^.Node;
               CurrentC^.Next := nil;
            end;
     end;  { of procedure Partition }



     begin  {________ BuildGSpaceWithRotation ________}
        reset(OBSTACLES);
        rewrite(ROTATION);

        rr := Pi / 6;
        hh := 2;
        vv := 1;

        CreateObject(Object);
        Partition(Object, ObjectA, ObjectB, ObjectC);
        RotationObjects(ObjectA, ObjectB, ObjectC, rr, hh, vv);

     end;  { of Procedure BuildGSpaceWithRotation }




begin   {______ Main ______}

   BuildGSpaceWithRotation;

end.    {___ Of  Main ___}
```

Appendix D: I/O FILES for the Rotational GSpace

[OBSTACLES]

3.0000	18.0000
9.0000	18.0000
9.0000	10.0000
3.0000	10.0000
10.5000	13.0000
19.0000	13.0000
19.0000	9.0000
10.5000	9.0000
8.0000	7.5000
16.0000	7.5000
16.0000	3.0000
8.0000	3.0000

[ROTATION]

0.0` Rotation

1.0000	18.0000
9.0000	18.0000
9.0000	9.0000
1.0000	9.0000
8.5000	13.0000
19.0000	13.0000
19.0000	8.0000
8.5000	8.0000
6.0000	7.5000
16.0000	7.5000
16.0000	2.0000
6.0000	2.0000

30.0` Rotation

1.2679	17.0000
3.0000	18.0000
9.0000	18.0000
9.5000	17.1340
9.5000	9.1340
7.7679	8.1340
1.7679	8.1340
1.2679	9.0000

8.7679	12.0000
10.5000	13.0000
19.0000	13.0000
19.5000	12.1340
19.5000	8.1340
17.7679	7.1340
9.2679	7.1340
8.7679	8.0000

6.2679	6.5000
8.0000	7.5000
16.0000	7.5000
16.5000	6.6340
16.5000	2.1340
14.7679	1.1340
6.7679	1.1340
6.2679	2.0000

60.0` Rotation

2.0000	16.2679
3.0000	18.0000
9.0000	18.0000
9.8660	17.5000
9.8660	9.5000
8.8660	7.7679
2.8660	7.7679
2.0000	8.2679

9.5000	11.2679
10.5000	13.0000
19.0000	13.0000
19.8660	12.5000
19.8660	8.5000
18.8660	6.7679
10.3660	6.7679
9.5000	7.2679

7.0000	5.7679
8.0000	7.5000
16.0000	7.5000
16.8660	7.0000
16.8660	2.5000
15.8660	0.7679
7.8660	0.7679
7.0000	1.2679

90.0` Rotation

2.0000	18.0000
3.0000	18.0000
9.0000	18.0000
9.0000	16.0000
9.0000	8.0000
8.0000	8.0000
2.0000	8.0000
2.0000	10.0000
9.5000	13.0000
10.5000	13.0000
19.0000	13.0000
19.0000	11.0000
19.0000	7.0000
18.0000	7.0000
9.5000	7.0000
9.5000	9.0000
7.0000	7.5000
8.0000	7.5000
16.0000	7.5000
16.0000	5.5000
16.0000	1.0000
15.0000	1.0000
7.0000	1.0000
7.0000	3.0000

120.0` Rotation

2.1340	17.5000
3.0000	18.0000
9.0000	18.0000
10.0000	16.2679
10.0000	8.2679
9.1340	7.7679
3.1340	7.7679
2.1340	9.5000
9.6340	12.5000
10.5000	13.0000
19.0000	13.0000
20.0000	11.2679
20.0000	7.2679
19.1340	6.7679
10.6340	6.7679
9.6340	8.5000

7.1340	7.0000
8.0000	7.5000
16.0000	7.5000
17.0000	5.7679
17.0000	1.2679
16.1340	0.7679
8.1340	0.7679
7.1340	2.5000

150.0` Rotation

2.5000	17.1340
3.0000	18.0000
9.0000	18.0000
10.7321	17.0000
10.7321	9.0000
10.2321	8.1340
4.2321	8.1340
2.5000	9.1340

10.0000	12.1340
10.5000	13.0000
19.0000	13.0000
20.7321	12.0000
20.7321	8.0000
20.2321	7.1340
11.7321	7.1340
10.0000	8.1340

7.5000	6.6340
8.0000	7.5000
16.0000	7.5000
17.7321	6.5000
17.7321	2.0000
17.2321	1.1340
9.2321	1.1340
7.5000	2.1340

Appendix E: Simulation of the Branch and Bound Algorithm

```
program BranchAndBoundAlgorithm(BBinput,BBoutput);
{____________________________________________________________

    Author  :  C. H. Chung

    Version :  2.0

     ate    :  November 1, 1988

 ___________________________________________________________
 NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
 This NodeSet is implemented by linked list, which node has
             the record structure to represent the vertices.
 Input file comes from BBinput.
 Output file is BBoutput.
 ____________________________________________________________}

type

   PointType = record
                  x, y, z : real
               end;

   NodeType = ^Nodes;
   Nodes = record
              Node : PointType;
              Next : NodeType
           end;

var

   BBinput, BBoutput : text;
   NodeSet, MinSet : NodeType;






   procedure PrintNodes(NodeSet : NodeType);
   {_____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
      This Procedure will print the NodeSet.
    _____________________________________________________________}
   begin
      if NodeSet = nil
         then
         else begin
                 writeln(BBoutput, NodeSet^.Node.x :10:4,
```

```
                                               NodeSet^.Node.y :10:4,
                                               NodeSet^.Node.z :10:4);
                  writeln(BBoutput);
                  PrintNodes(NodeSet^.Next)
               end
   end;






   function EuclideanDistance(NodeSet : NodeType) : real;
   {____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
      This Function will calculate the Euclidean Distance
                       between the points in 3D.
    ____________________________________________________________}
   var
      Current : NodeType;
      d1, d2, d3 : real;
      x1, y1, z1,
      x2, y2, z2 : real;
   begin
      Current := NodeSet;
      x1 := Current^.Node.x;
      y1 := Current^.Node.y;
      z1 := Current^.Node.z;

      Current := Current^.Next;
      x2 := Current^.Node.x;
      y2 := Current^.Node.y;
      z2 := Current^.Node.z;

      d1 := (x2 - x1) * (x2 - x1);
      d2 := (y2 - y1) * (y2 - y1);
      d3 := (z2 - z1) * (z2 - z1);

      EuclideanDistance := sqrt(d1 + d2 + d3)
   end;






   function LengthOfNodeSet(NodeSet : NodeType) : integer;
   {____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
      This Function will find the length of NodeSet.
```

```
    ____________________________________________________________}
begin
   if (NodeSet^.Next = nil)
      then LengthOfNodeSet := 1
      else LengthOfNodeSet := 1 + LengthOfNodeSet(NodeSet^.Next)
end;







function Distance(NodeSet : NodeType) : real;
{____________________________________________________________
   NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
   # of NodeSet for Distance >= 2
   Function EuclideanDistance will find the Euclidean Distance
               between the fist node and the second in NodeSet.
 ____________________________________________________________}
var
   Current : NodeType;
begin
   Current := NodeSet;
   if (LengthOfNodeSet(Current) <= 2)
      then Distance := EuclideanDistance(Current)
      else Distance := EuclideanDistance(Current)
                         + Distance(Current^.Next)
end;






procedure Copy(var NodeSet : NodeType; var MinSet : NodeType);
{____________________________________________________________
   NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
   This Procedure will duplicate the NodeSet in the other memory
                                                        storage.
 ____________________________________________________________}
var
   NodeHolder, Current : NodeType;

begin
   MinSet := nil;
   NodeHolder := NodeSet;

   new(MinSet);
   MinSet^.Node := NodeSet^.Node;
   MinSet^.Next := nil;
```

```
         Current := MinSet;

         while (NodeSet^.Next <> nil)
            do begin
                  new(Current^.Next);
                  Current := Current^.Next;
                  NodeSet := NodeSet^.Next;
                  Current^.Node := NodeSet^.Node;
                  Current^.Next := nil;
               end;
         NodeSet := NodeHolder;
      end;




      procedure BranchAndBound(var NodeSet : NodeType;
                               var MinSet : NodeType);
      {_______________________________________________________________
         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
         This Procedure will find the the compensated nodes by
                                            Branch and Bound Method.
       _______________________________________________________________}

      var
         N1, N2, N2holder,
         Increament, MinDistance : real;
         Head: NodeType;

      begin
         Increament := 0.01;
         Head := NodeSet;
         Copy(NodeSet,MinSet);

         NodeSet := NodeSet^.Next;
         N1 := NodeSet^.Node.z;
         NodeSet := NodeSet^.Next;
         N2 := NodeSet^.Node.z;
         N2holder := N2;
         NodeSet := Head;

         MinDistance := Distance(NodeSet);

         while (N1 > 0.0)
            do begin
                  while (N2 > 0.0)
                     do begin
                           N2 := N2 - Increament;
                           Head := NodeSet;
```

```
                              NodeSet := NodeSet^.Next;
                              NodeSet := NodeSet^.Next;
                              NodeSet^.Node.z := N2;
                              NodeSet := Head;

                              if (MinDistance > Distance(NodeSet))
                                 then begin
                                         MinDistance := Distance(NodeSe
                                         Copy(NodeSet,MinSet);
                                      end;
                           end;

                     N2 := N2holder;

                     N1 := N1 - Increament;
                     Head := NodeSet;
                     NodeSet := NodeSet^.Next;
                     NodeSet^.Node.z := N1;
                     NodeSet := Head;
                  end;
      end;

procedure CreateNodes(var NodeSet : NodeType);
{__________________________________________________________________
   NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
   This Procedure will create the NodeSet.
 __________________________________________________________________}
var
   Current : NodeType;
begin
   NodeSet := nil;
   if not eof(BBinput)
      then begin
              new(NodeSet);
              readln(BBinput, NodeSet^.Node.x,
                              NodeSet^.Node.y,
                              NodeSet^.Node.z);
              NodeSet^.Next := nil;
              Current := NodeSet;
              while not eof(BBinput)
                 do begin
                       new(Current^.Next);
                       Current := Current^.Next;
                       readln(BBinput, Current^.Node.x,
                                       Current^.Node.y,
```

```
                                                            Current^.Node.z);
                                            Current^.Next := nil
                                        end
                             end
     end;




begin {_____ MAIN _____}
     reset(BBinput);
     rewrite(BBoutput);

     CreateNodes(NodeSet);
     writeln(BBoutput);
     writeln(BBoutput);
     writeln(BBoutput);
     writeln(BBoutput, 'The original vertices by VGraph Algorithm');
     writeln(BBoutput);
     writeln(BBoutput, '________________________ orginal distance = ',
                                                       Distance(NodeSet) :7:4);
     PrintNodes(NodeSet);
     BranchAndBound(NodeSet,MinSet);
     writeln(BBoutput);
     writeln(BBoutput);
     writeln(BBoutput);
     writeln(BBoutput, 'The vertices compensated by BranchAndBound');
     writeln(BBoutput, '_____________________________',
                                                          Distance(MinSet) :10:4);
     PrintNodes(MinSet);
     writeln(BBoutput);
     writeln(BBoutput);
     writeln(BBoutput,'          clock  = ', clock);
     writeln(BBoutput,'          system = ', syscl...

end. {_____ MAIN _____}
```

Appendix F: I/O FILES for the Branch and Bound Algorithm

[BBinput]

```
3  2  4
7  4  10
8  8  9
4  11 2
```

[BBoutput]

```
The original vertices by VGraph Algorithm

________________________________ orginal distance = 20.3283
     3.0000      2.0000      4.0000

     7.0000      4.0000     10.0000

     8.0000      8.0000      9.0000

     4.0000     11.0000      2.0000


The vertices compensated by BranchAndBound
_______________________________________ 13.7416
     3.0000      2.0000      4.0000

     7.0000      4.0000      3.3400

     8.0000      8.0000      2.7300

     4.0000     11.0000      2.0000


            clock  =     2301666
            system =       10116
```

Appendix G: Simulation of the RCA

```
program RCAlgorithm(RCAinput,RCAoutput);
{_______________________________________________________________

     Author   :  C. H. Chung

     Version  :  2.0

     Date     :  October 25, 1988

     ____________________________________________________________
     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by th
     This NodeSet is implemented by linked list, which node has
               the record structure to represent the vertices.
     Input file comes from RCAinput.
     Output file is RCAoutput.
     Determine Error to decide the accuracy.
     Refer to Appendix C in RAL-TR-88-117.
type ____________________________________________________________}

   PointType = record
                   x, y, z : real
               end;

   NodeType = ^Nodes;
   Nodes = record
              Node : PointType;
              Next : NodeType
           end;

var

   RCAinput, RCAoutput : text;
   NodeSet : NodeType;
   Error : real;






   procedure PrintNodes(NodeSet : NodeType);
   {____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the Grap...
      This Procedure will print the NodeSet.
   ____________________________________________________________}
   begin
      if NodeSet = nil
        then
```

```
            else begin
                    writeln(RCAoutput, NodeSet^.Node.x :10:4,
                                       NodeSet^.Node.y :10:4,
                                       NodeSet^.Node.z :10:4);
                    writeln(RCAoutput);
                    PrintNodes(NodeSet^.Next)
                 end
      end;





      function EuclideanDistance(NodeSet : NodeType) : real;
      {_____________________________________________________________
         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
         This Function will calculate the Euclidean Distance
                                          between the points in 3D.
       _____________________________________________________________}
      var
         Current : NodeType;
         d1, d2, d3 : real;
         x1, y1, z1,
         x2, y2, z2 : real;
      begin
         Current := NodeSet;
         x1 := Current^.Node.x;
         y1 := Current^.Node.y;
         z1 := Current^.Node.z;

         Current := Current^.Next;
         x2 := Current^.Node.x;
         y2 := Current^.Node.y;
         z2 := Current^.Node.z;

         d1 := (x2 - x1) * (x2 - x1);
         d2 := (y2 - y1) * (y2 - y1);
         d3 := (z2 - z1) * (z2 - z1);

         EuclideanDistance := sqrt(d1 + d2 + d3)
      end;



      procedure Reset(var NodeSet : NodeType);
      {_____________________________________________________________
         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
         Refer to Appendix C in RAL-TR-88-117.
       _____________________________________________________________}
```

```
var
   Current : NodeType;
   x0, y0, z0,
   x1, y1, z1,
   x2, y2, z2,
   c1, c2, c3, c4,
   p, q, r,
   d1, d2, ed1, ed2 : real;
begin
   Current := NodeSet;
   x0 := Current^.Node.x;
   y0 := Current^.Node.y;
   z0 := Current^.Node.z;

   Current := Current^.Next;
   x1 := Current^.Node.x;
   y1 := Current^.Node.y;
   z1 := Current^.Node.z;

   Current := Current^.Next;
   x2 := Current^.Node.x;
   y2 := Current^.Node.y;
   z2 := Current^.Node.z;

   c1 := (x1-x0)*(x1-x0)  +  (y1-y0)*(y1-y0);
   c2 := z2-z1;
   c3 := (x2-x1)*(x2-x1)  +  (y2-y1)*(y2-y1);
   c4 := z1-z0;

   p := c1-c3;
   q := 2 *(c1*c2 + c3*c4);
   r := c1*c2*c2 - c3*c4*c4;

   if (p = 0)
      then d1 := -r / q
      else begin
              d1 := (-q + sqrt(q*q - 4*p*r))/(2*p
              d2 := (-q - sqrt(q*q - 4*p*r))/(2*p
              ed1 :=   sqrt((x1-x0)  *  (x1-x0)
                           +(y1-y0)  *  (y1-y0)
                           +(z1-z0-d1)  *  (z1-z0-d1))
                     + sqrt((x2-x1)  *  (x2-x1)
                           +(y2-y1)  *  (y2-y1)
                           +(z2-z1+d1)  *  (z2-z1+d1));
              ed2 :=   sqrt((x1-x0)  *  (x1-x0)
                           +(y1-y0)  *  (y1-y0)
                           +(z1-z0-d2)  *  (z1-z0-d2))
                     + sqrt((x2-x1)  *  (x2-x1)
                           +(y2-y1)  *  (y2-y1)
                           +(z2-z1+d2)  *  (z2-z1+d2));
```

```
                if (ed1 > ed2)
                   then d1 := d2
             end;

      NodeSet^.Next^.Node.z := z1 - d1
   end;




   function LengthOfNodeSet(NodeSet : NodeType) : integer;
   {_____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
      This Function will find the length of NodeSet.
    _____________________________________________________________}
   begin
      if (NodeSet^.Next = nil)
         then LengthOfNodeSet := 1
         else LengthOfNodeSet := 1 + LengthOfNodeSet(NodeSet^.Next)
   end;




   procedure Compensate(var NodeSet : NodeType);
   {_____________________________________________________________
      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
      # of NodeSet for Compensate >= 3.
      Procedure Reset will take the first 3 nodes in NodeSet
                replace the second of the 3 nodes in NodeSet in
                order to get the set of the compensated nodes
    _____________________________________________________________
   begin
      if (LengthOfNodeSet(NodeSet) = 3)
         then Reset(NodeSet)
         else begin
                 Reset(NodeSet);
                 Compensate(NodeSet^.Next)
              end
   end;




   function Distance(NodeSet : NodeType) : real;
   {_____________________________________________________________
```

```
    NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
    # of NodeSet for Distance >= 2
    Function EuclideanDistance will find the Euclidean Distance
             between the fist node and the second in NodeSet.
  ____________________________________________________________}
 var
    Current : NodeType;
 begin
    Current := NodeSet;
    if (LengthOfNodeSet(Current) <= 2)
       then Distance := EuclideanDistance(Current)
       else Distance := EuclideanDistance(Current)
                           + Distance(Current^.Next)
 end;





 procedure RCA(var NodeSet : NodeType; Error : real);
 {____________________________________________________________
    NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
    Error = a permissible error
    Function Distance will calculate the Euclidean Distance
                                              through NodeSet.
    Procedure Compensate will find the compensated nodes and
                            will return the set of these nodes.
  ____________________________________________________________
 var
    Path1, Path2 : real;
 begin
    Path1 := Distance(NodeSet);
    Compensate(NodeSet);
    Path2 := Distance(NodeSet);
    writeln(RCAoutput,'_________________ compensa[illegible]
                                              Path [illegible]
    PrintNodes(NodeSet);
    if (abs(Path1 - Path2) > Error)
       then RCA(NodeSet, Error)
 end;





 procedure CreateNodes(var NodeSet : NodeType);
 {____________________________________________________________
    NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
```

```
      This Procedure will create the NodeSet.
    _____________________________________________________________}
    var
      Current : NodeType;
    begin
      NodeSet := nil;
      if not eof(RCAinput)
         then begin
                 new(NodeSet);
                 readln(RCAinput, NodeSet^.Node.x,
                                  NodeSet^.Node.y,
                                  NodeSet^.Node.z);
                 NodeSet^.Next := nil;
                 Current := NodeSet;
                 while not eof(RCAinput)
                    do begin
                          new(Current^.Next);
                          Current := Current^.Next;
                          readln(RCAinput, Current^.Node.x,
                                           Current^.Node.y,
                                           Current^.Node.z);
                          Current^.Next := nil
                       end
              end
    end;





  begin {_____   MAIN   _____}
    reset(RCAinput);
    rewrite(RCAoutput);

    Error := 0.00001;
    CreateNodes(NodeSet);
    writeln(RCAoutput);
    writeln(RCAoutput);
    writeln(RCAoutput);
    writeln(RCAoutput, 'The original vertices by VGraph Algorithm');
    writeln(RCAoutput);
    writeln(RCAoutput, '______________________ orginal distance = ',
                                                Distance(NodeSet) :7:4);
    PrintNodes(NodeSet);
    RCA(NodeSet, Error);
    writeln(RCAoutput);
    writeln(RCAoutput);
    writeln(RCAoutput);
    writeln(RCAoutput, 'The vertices compensated by RCA');
```

```
    writeln(RCAoutput, '________________________________________');
    PrintNodes(NodeSet);
    writeln(RCAoutput);
    writeln(RCAoutput);
    writeln(RCAoutput,'          clock  = ', clock);
    writeln(RCAoutput,'          system = ', sysclock);
end. {_____   MAIN   _____}
```

Appendix H: I/O FILES for the RCA

[RCAinput]

```
3  2  4
7  4  10
8  8  9
4  11 2
```

[RCAoutput]

The original vertices by VGraph Algorithm

orginal distance = 20.3283

3.0000	2.0000	4.0000
7.0000	4.0000	10.0000
8.0000	8.0000	9.0000
4.0000	11.0000	2.0000

compensated distance = 15.3916

3.0000	2.0000	4.0000
7.0000	4.0000	6.6015
8.0000	8.0000	4.5219
4.0000	11.0000	2.0000

compensated distance = [illegible]

3.0000	2.0000	4.0000
7.0000	4.0000	4.2715
8.0000	8.0000	3.2449
4.0000	11.0000	2.0000

compensated distance = 13.7529

3.0000	2.0000	4.0000
7.0000	4.0000	3.6071
8.0000	8.0000	2.8808
4.0000	11.0000	2.0000

compensated distance = 13.7425

3.0000	2.0000	4.0000
7.0000	4.0000	3.4177
8.0000	8.0000	2.7770
4.0000	11.0000	2.0000

compensated distance = 13.7416

3.0000	2.0000	4.0000
7.0000	4.0000	3.3637
8.0000	8.0000	2.7474
4.0000	11.0000	2.0000

compensated distance = 13.7416

3.0000	2.0000	4.0000
7.0000	4.0000	3.3482
8.0000	8.0000	2.7389
4.0000	11.0000	2.0000

compensated distance = 13.7416

3.0000	2.0000	4.0000
7.0000	4.0000	3.3439
8.0000	8.0000	2.7365
4.0000	11.0000	2.0000

The vertices compensated by RCA

3.0000	2.0000	4.0000
7.0000	4.0000	3.3439
8.0000	8.0000	2.7365
4.0000	11.0000	2.0000

```
clock  =        400
system =         83
```

Appendix I: Simulation of the OPM

```
program OrthogonalProjectionMethod(OBJECT, PROJECTION);
{______________________________________________________________

    Author  :  C. H. Chung

    Version :  2.3

    Date    :  December 3, 1988

 ______________________________________________________________

    OPM;

          .  To build the Grown Space Obstacles in 3D by OPM
          .  INPUT  FILE : OBJECT
          .  OUTPUT FILE : PROJECTION


    This program will build the Grown Space Obstacles in 3D.
 ______________________________________________________________}
type
    Point2D = record
                    x, y : real;
                end;
    Vertice2D = ^Node2D;
    Node2D = record
                  Node : Point2D;
                  Next : Vertice2D
              end;

    Point3D = record
                    x, y, z : real;
                end;
    Vertice3D = ^Node3D;
    Node3D = record
                  Node : Point3D;
                  Next : Vertice3D
              end;
var
    OBJECT, PROJECTION : text;
    LinkedVertices : Vertice3D;





     procedure OPM(var LinkedVertices : Vertice3D);
     {_________________________________________________________

        Author  :  C. H. Chung
```

```
    Version :  2.3

    Date    :  December 3, 1988

  ______________________________________________________________

    OPM(LinkedVertices);

       . To build the Grown Space Obstacles in 3D by OPM.
       . INPUT  FILE : OBJECT
       . OUTPUT FILE : PROJECTION


    This program will build the Grown Space Obstacles.

    hh : the horizontal length of the object

    vv : the vertical length of the object

    rr : the sliced angle for rotational Grown Space Obstacles.
  ______________________________________________________________}
const
   Pi = 3.141592;     {Radian}
var
   Object : Vertice3D;
   ObjectXY, ObjectYZ, ObjectXZ : Vertice2D;
   GrownXY, GrownYZ, GrownXZ : Vertice2D;
   rr : real;
   hh1, hh2, hh3,
   vv1, vv2, vv3 : real;








procedure Print2Dvertice(List : Vertice2D);
{________________________________________________________
   This procedure will print the Linked List of the
   shortest path.
_________________________________________________________}
begin
   if List = nil
      then writeln(PROJECTION)
      else begin
              writeln(PROJECTION, List^.Node.x :10:4,
                                  List^.Node.y :10:4);
              Print2Dvertice(List^.Next)
```

```
                end
    end;







    procedure Print3Dvertice(List : Vertice3D);
    {______________________________________________________________
       This procedure will print the Linked List of the
       shortest path.
    ______________________________________________________________}
    begin
       if List = nil
          then writeln(PROJECTION)
          else begin
                  writeln(PROJECTION, List^.Node.x :10:4,
                                      List^.Node.y :10:4,
                                      List^.Node.z :10:4);
                  Print3Dvertice(List^.Next)
               end
    end;






    procedure CreateObject(var Object : Vertice3D);
    {______________________________________________________________
       This procedure creates the object from the input file
       by the linked list.
    ______________________________________________________________}
    var
       Current : Vertice3D;
    begin
       Object := nil;
       if not eof(OBJECT)
          then begin
                  new(Object);
                  readln(OBJECT, Object^.Node.x,
                                 Object^.Node.y,
                                 Object^.Node.z);
                  Object^.Next := nil;
                  Current := Object;
                  while not eof(OBJECT)
                     do begin
                           new(Current^.Next);
```

```
                              Current := Current^.Next;
                              readln(OBJECT, Current^.Node.x,
                                             Current^.Node.y,
                                             Current^.Node.z);
                              Current^.Next := nil
                          end
                  end
end;






procedure GrownObject(var Object, Grown : Vertice2D;
                         hh, vv,  rr : real);
{_______________________________________________________________
    This procedure builds the Grown Space Obstacles.

                                         where h : horizonta
                                               v : vertica

    0 < q < Pi/2

         a1  =  (A1x,A1y)  + h(-cos(q),-sin(q))
         a2  =  (A1x,A1y)
         a3  =  (A2x,A2y)
         a4  =  (A2x,A2y)  + v(sin(q),-cos(q))
         a5  =  (A3x,A3y)  + v(sin(q),-cos(q))
         a6  =  (a5x,a5y)  + h(-cos(q),-sin(q))
         a8  =  (A4x,A4y)  + h(-cos(q),-sin(q))
         a7  =  (a8x,a8y)  + v(sin(q),-cos(q))

    Pi/2 < q < Pi

         q = q - Pi/2
         temp = h      (to swap h and v)
         h = v
         v = temp

    q = 0
         Delete a2, a4, a6, a8.

    q = Pi/2
         Swap h and v.
         Delete a2, a4, a6, a8.
  _______________________________________________________________}
 var
    Current, Head : Vertice2D;
```

```
      begin
         Current := nil;
         new(Current);
         Current^.Node.x := Object^.Node.x - hh * cos(rr);
         Current^.Node.y := Object^.Node.y - hh * sin(rr);
         Current^.Next := nil;

         Head := Object;
         Grown := Current;

         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x;
         Current^.Node.y := Object^.Node.y;
         Current^.Next := nil;

         Object := Object^.Next;
         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x;
         Current^.Node.y := Object^.Node.y;
         Current^.Next := nil;

         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x + vv * sin(rr);
         Current^.Node.y := Object^.Node.y - vv * cos(rr);
         Current^.Next := nil;

         Object := Object^.Next;
         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x + vv * sin(rr);
         Current^.Node.y := Object^.Node.y - vv * cos(rr);
         Current^.Next := nil;

         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x + vv*sin(rr) - hh*cos(rr);
         Current^.Node.y := Object^.Node.y - vv*cos(rr) - hh*sin(rr);
         Current^.Next := nil;

         Object := Object^.Next;
         new(Current^.Next);
         Current := Current^.Next;
         Current^.Node.x := Object^.Node.x + vv*sin(rr) - hh*cos(rr);
         Current^.Node.y := Object^.Node.y - vv*cos(rr) - hh*sin(rr);
         Current^.Next := nil;

         new(Current^.Next);
```

```
      Current := Current^.Next;
      Current^.Node.x := Object^.Node.x - hh * cos(rr);
      Current^.Node.y := Object^.Node.y - hh * sin(rr);
      Current^.Next := nil;

      Object := Head;

      if (rr = 0) or (abs(rr - Pi/2) < 0.001)
         then begin
                 Current := nil;
                 new(Current);
                 Current^.Node := Grown^.Node;
                 Current^.Next := nil;

                 Head := nil;
                 Head := Current;
                 while Grown^.Next^.Next <> nil
                    do begin
                          new(Current^.Next);
                          Current := Current^.Next;
                          Grown := Grown^.Next^.Next,
                          Current^.Node := Grown^.Node;
                          Current^.Next := nil;
                       end;
                Grown := Head;
             end;
   end;






   procedure OrthogonalProjection(Object : Vertice3D;
                   var ObjectXY, ObjectYZ, ObjectXZ : Vertice2D);
   {______________________________________________________________
      This procedure project Object in 3D into 3 Objects in 2D.
    ______________________________________________________________}
   var
      Current : Vertice3D;
      CurrentXY,
      CurrentYZ,
      CurrentXZ : Vertice2D;
      X1, X2,
      Y1, Y2,
      Z1, Z2 : real;

   begin

      Current := Object;
```

```
ObjectXY := nil;
ObjectYZ := nil;
ObjectXZ := nil;

X1 := Current^.Node.x;
Y1 := Current^.Node.y;
Z1 := Current^.Node.z;

Current := Current^.Next;
Y2 := Current^.Node.y;

Current := Current^.Next;
X2 := Current^.Node.x;

Current := Current^.Next;
Current := Current^.Next;
Z2 := Current^.Node.z;

new(ObjectXY);
ObjectXY^.Node.x := X1;
ObjectXY^.Node.y := Y2;
ObjectXY^.Next := nil;

new(ObjectYZ);
ObjectYZ^.Node.x := Y1;
ObjectYZ^.Node.y := Z2;
ObjectYZ^.Next := nil;

new(ObjectXZ);
ObjectXZ^.Node.x := X1;
ObjectXZ^.Node.y := Z2;
ObjectXZ^.Next := nil;

CurrentXY := ObjectXY;
CurrentYZ := ObjectYZ;
CurrentXZ := ObjectXZ;

new(CurrentXY^.Next);
CurrentXY := CurrentXY^.Next;
CurrentXY^.Node.x := X2;
CurrentXY^.Node.y := Y2;
CurrentXY^.Next := nil;

new(CurrentYZ^.Next);
CurrentYZ := CurrentYZ^.Next;
CurrentYZ^.Node.x := Y2;
CurrentYZ^.Node.y := Z2;
CurrentYZ^.Next := nil;
```

```
    new(CurrentXZ^.Next);
    CurrentXZ := CurrentXZ^.Next;
    CurrentXZ^.Node.x := X2;
    CurrentXZ^.Node.y := Z2;
    CurrentXZ^.Next := nil;

    new(CurrentXY^.Next);
    CurrentXY := CurrentXY^.Next;
    CurrentXY^.Node.x := X2;
    CurrentXY^.Node.y := Y1;
    CurrentXY^.Next := nil;

    new(CurrentYZ^.Next);
    CurrentYZ := CurrentYZ^.Next;
    CurrentYZ^.Node.x := Y2;
    CurrentYZ^.Node.y := Z1;
    CurrentYZ^.Next := nil;

    new(CurrentXZ^.Next);
    CurrentXZ := CurrentXZ^.Next;
    CurrentXZ^.Node.x := X2;
    CurrentXZ^.Node.y := Z1;
    CurrentXZ^.Next := nil;

    new(CurrentXY^.Next);
    CurrentXY := CurrentXY^.Next;
    CurrentXY^.Node.x := X1;
    CurrentXY^.Node.y := Y1;
    CurrentXY^.Next := nil;

    new(CurrentYZ^.Next);
    CurrentYZ := CurrentYZ^.Next;
    CurrentYZ^.Node.x := Y1;
    CurrentYZ^.Node.y := Z1;
    CurrentYZ^.Next := nil;

    new(CurrentXZ^.Next);
    CurrentXZ := CurrentXZ^.Next;
    CurrentXZ^.Node.x := X1;
    CurrentXZ^.Node.y := Z1;
    CurrentXZ^.Next := nil;
  end;






  procedure ReconstructObject(ObjectXY, ObjectYZ, ObjectXZ :
                       Vertice2D; var LinkedVertices : Vertice3D);
```

```
{______________________________________________________________
    This procedure will reconstruct the object from the
    3 projected images in 2D.
______________________________________________________________}
var
   Current : Vertice3D;
   CurrentXY,
   CurrentYZ,
   CurrentXZ : Vertice2D;
   X1, Y1, Z1,
   X2, Y2, Z2 : real;
begin
   LinkedVertices := nil;
   new(LinkedVertices);
   Current := LinkedVertices;
   LinkedVertices^.Next := nil;

   CurrentXY := ObjectXY;
   CurrentYZ := ObjectYZ;
   CurrentXZ := ObjectXZ;

   while CurrentXY <> nil do
      begin
         X1 := CurrentXY^.Node.x;
         Y1 := CurrentXY^.Node.y;
         CurrentXY := CurrentXY^.Next;
         while CurrentYZ <> nil do
            begin
               Y2 := CurrentYZ^.Node.x;
               Z1 := CurrentYZ^.Node.y;
               CurrentYZ := CurrentYZ^.Next;
               if (Y1 = Y2)
                  then while CurrentXZ <> nil do
                          begin
                             X2 := CurrentXZ^.Node.x;
                             Z2 := CurrentXZ^.Node.y;
                             CurrentXZ := CurrentXZ^.Next;
                             if (Z1 = Z2) and (X1 = X2)
                                then begin
                                        Current^.Node.x := X1;
                                        Current^.Node.y := Y1;
                                        Current^.Node.z := Z1;
                                        Current^.Next := nil;
                                        if (CurrentXY = nil) and
                                           (CurrentYZ = nil) and
                                           (CurrentXZ = nil)
                                           then
                                           else begin
                                        new(Current^.Next);
```

```
                                                Current := Current^.Next;
                                                      end;
                                      end;
                              end;
                    CurrentXZ := ObjectXZ;
                 end;
              CurrentYZ := ObjectYZ;
           end;
        Current := LinkedVertices;
        while Current^.Next^.Next <> nil
           do Current := Current^.Next;
        Current := nil;
     end;



     begin   {_______ OPM ________}
        reset(OBJECT);
        rewrite(PROJECTION);

        rr := Pi/6;
        rr := 0;
        hh1 := 1.0;
        hh2 := 1.5;
        hh3 := 1.0;
        vv1 := 1.5;
        vv2 := 0.5;
        vv3 := 0.5;

        CreateObject(Object);
        writeln(PROJECTION, ' Object in 3D:');
        writeln(PROJECTION);
        Print3Dvertice(Object);

        OrthogonalProjection(Object, ObjectXY, ObjectYZ, ObjectXZ);
        writeln(PROJECTION);
        writeln(PROJECTION);
        writeln(PROJECTION, ' Image projected in 2D:');
        writeln(PROJECTION);
        writeln(PROJECTION, '     (X,Y) projection');
        Print2Dvertice(ObjectXY);
        writeln(PROJECTION);
        writeln(PROJECTION, '     (Y,Z) projection');
        Print2Dvertice(ObjectYZ);
        writeln(PROJECTION);
        writeln(PROJECTION, '     (X,Z) projection');
        Print2Dvertice(ObjectXZ);

        GrownObject(ObjectXY, GrownXY, hh1, vv1, rr);
        GrownObject(ObjectYZ, GrownYZ, hh2, vv2, rr);
```

```
        GrownObject(ObjectXZ, GrownXZ, hh3, vv3, rr);

        writeln(PROJECTION);
        writeln(PROJECTION);
        writeln(PROJECTION, ' Grown Image projected in 2D:');
        writeln(PROJECTION);
        writeln(PROJECTION);
        writeln(PROJECTION, '    (X,Y) Grown Image');
        Print2Dvertice(GrownXY);
        writeln(PROJECTION);
        writeln(PROJECTION, '    (Y,Z) Grown Image');
        Print2Dvertice(GrownYZ);
        writeln(PROJECTION);
        writeln(PROJECTION, '    (X,Z) Grown Image');
        Print2Dvertice(GrownXZ);

        ReconstructObject(GrownXY, GrownYZ, GrownXZ, LinkedVertices);
        writeln(PROJECTION);
        writeln(PROJECTION);
        writeln(PROJECTION, ' Object in 3D reconstructed [illegible]');
        writeln(PROJECTION);
        Print3Dvertice(LinkedVertices);

    end;  { of Procedure OPM }




begin  {___ Main ___}
    OPM(LinkedVertices);
end.   {__ of Main __}
```

Appendix J: I/O FILES for the OPM

[OBJECT]

7	5	3
7	10	3
14	10	3
14	5	3
7	5	12
7	10	12
14	10	12
14	5	12

[PROJECTION]

Object in 3D:

7.0000	5.0000	3.0000
7.0000	10.0000	3.0000
14.0000	10.0000	3.0000
14.0000	5.0000	3.0000
7.0000	5.0000	12.0000
7.0000	10.0000	12.0000
14.0000	10.0000	12.0000
14.0000	5.0000	12.0000

Image projected in 2D:

(X,Y) projection

7.0000	10.0000
14.0000	10.0000
14.0000	5.0000
7.0000	5.0000

(Y,Z) projection

5.0000	12.0000
10.0000	12.0000
10.0000	3.0000
5.0000	3.0000

(X,Z) projection	
7.0000	12.0000
14.0000	12.0000
14.0000	3.0000
7.0000	3.0000

Grown Image projected in 2D:

(X,Y) Grown Image	
6.0000	10.0000
14.0000	10.0000
14.0000	3.5000
6.0000	3.5000

(Y,Z) Grown Image	
3.5000	12.0000
10.0000	12.0000
10.0000	2.5000
3.5000	2.5000

(X,Z) Grown Image	
6.0000	12.0000
14.0000	12.0000
14.0000	2.5000
6.0000	2.5000

Object in 3D reconstructed by OPM:

6.0000	10.0000	12.0000
6.0000	10.0000	2.5000
14.0000	10.0000	12.0000
14.0000	10.0000	2.5000
14.0000	3.5000	12.0000
14.0000	3.5000	2.5000
6.0000	3.5000	12.0000
6.0000	3.5000	2.5000